Software Communications Architecture Specification

MSRC-5000SCA
V2.1
13 July, 2001

Prepared for the
Joint Tactical Radio System (JTRS) Joint Program Office

Prepared by the
Modular Software-programmable Radio Consortium
under Contract No. DAAB15-00-3-0001

Revision Summary

1.0	Formal release for initial validation.
1.1	Incorporate approved Change Proposals, numbers 97, 99, 110, 160, 161, 162, 164, 171, 177, 178, 179, 180, 193, 195, 201, 204, 205, 208, 209, 211, 216.
2.0	Incorporate approved Change Proposals, numbers 39, 105, 119, 147, 175, 186, 191, 192, 210, 217, 218, 219, 220, 222, 223, 225, 226, 227, 229, 231, 232, 235, 237, 240, 243, 249, 255, 258, 266, 270, 275, 276, 277, 278, 282, 283, 285, 299, 307, 308, 310, 311, 332, 335, 336, 337, 341, 342, 343, 344, 345.
2.1	Incorporate approved Change Proposals, numbers 88, 102, 142, 306, 316, 353, 357, 358, 359, 360, 365, 366, 367, 369, 370, 371, 372, 373, 419, 468, 471, 472, 473, 475, 476, 477

Changes from the previous revision, other than editorial corrections, are marked with change bars in the margins.

Change Proposals are controlled by the JTRS Change Control Board. CPs incorporated into the SCA are considered "closed" and can be seen on the JTRS web site at: www.jtrs.sarda.army.mil/docs/documents/sca_ccb.html.

Table of Contents

List of Figures

Foreword

Introduction. The Software Communication Architecture (SCA) specification is published by the Joint Tactical Radio System (JTRS) Joint Program Office (JPO). This program office was established to pursue the development of future communication systems, capturing the benefits of the technology advances of recent years which are expected to greatly enhance interoperability of communication systems and reduce development and deployment costs. The goals set for the JTRS program are:

- Greatly increased operational flexibility and interoperability of globally deployed systems,
- Reduced supportability costs,
- Upgradeability in terms of easy technology insertion and capability upgrades, and
- Reduced system acquisition and operation cost.

In order to achieve these goals, the SCA has been structured to

- provide for portability of applications software between different SCA implementations,
- leverage commercial standards to reduce development cost,
- reduce development time of new waveforms through the ability to reuse design modules, and
- build on evolving commercial frameworks and architectures.

The SCA is deliberately designed to meet commercial application requirements as well as military applications. It is the expectation of the Government that the basic SCA will become a commercially approved standard. It is for this reason that a wide cross section of industry has been invited to participate in the development and the validation of the SCA. The SCA is not a system specification, as it is intended to be implementation independent, but a set of rules that constrain the design of systems to achieve the objectives listed above. The SCA specification version 1.0 established the baseline for architecture validation. The validation effort demonstrated that multiple vendors could independently design systems, which, when built according to the SCA requirements, meet the program goals outlined above. Lessons learned during the validation have been incorporated in SCA version 2.0.

The SCA documentation consists of the basic architecture specification, a supplement on military security, a supplement on definition of application program interfaces, and a rationale document.

Software Structure. The software framework of the SCA defines the Operating Environment (OE) and specifies the services and interfaces that applications use from that environment. The OE is comprised of:

- a Core Framework (CF),
- a CORBA middleware, and
- a POSIX-based Operating System (OS) with associated board support packages.

The OE imposes design constraints on waveform and other applications to provide increased portability of those applications from one SCA-compliant radio platform to another. These design constraints include specified interfaces between the Core Framework and application software, and restrictions on waveform usage of the Operating System.

The SCA also provides a building-block structure (defined in the API Supplement) for defining application programming interfaces (APIs) between application software components. This building-block structure for API definition facilitates component-level reuse and allows significant flexibility for developers to define waveform-specific APIs.

Core Framework. The CF is an architectural concept defining the essential, "core" set of open software Interfaces and Profiles that provide for the deployment, management, interconnection, and intercommunication of software application components in embedded, distributed-computing communication systems. All interfaces defined in section 3.1.3 of the SCA Specification are part of the CF. Some of them are implemented by Core Application Services developers; some are implemented by non-core Applications (i.e. waveforms, etc.); and some implemented by hardware device providers. The CF builds an information base from the collection of profiles, known as the Domain Profile and provided with the hardware and software of the system.

Hardware Structure. The hardware framework also uses OO concepts to define typical hardware partitions within realizable systems. The primary purpose of the hardware structure is to require complete and comprehensive publication of interfaces and attributes once systems have been built. With these published specifications, additional venders can provide modules within a system and software developers can identify hardware modules with capabilities required for a particular waveform application. Hardware modularity also facilitates technology insertion as future programmable elements increase in capability.

Military Applications. To maximize the commercial application of the SCA and the resulting benefit, military-unique requirements are provided in SCA supplements. Currently there are two supplements to the SCA Specification:

- a Security Supplement identifies requirements to insure adequate protection of military secure communications and to facilitate certification of JTRS products by the NSA, and
- an API Supplement identifies structures associated with radio system services at various interfaces such as physical, networking, security, and external interfaces. These APIs, when fully defined, improve portability of applications within JTRS implementations, and make reuse of functional components of those applications easier. For example, standardizing APIs for a security module within a JTRS enables reuse of common modules for multiple waveform applications. Standardizing networking APIs improves portability of networking applications and offers easier internetworking functions such as routing, bridging and providing gateways.

Support and Rationale Document (SRD). This document provides the rationale behind architectural decisions along with further supporting material.

Future Directions. The JTRS JPO intends to maintain the SCA Specification and Supplements over the next year. The goal of the JPO is to transition maintenance of the SCA to a commercial open-standards organization. Changes to the SCA will be incorporated based upon lessons-learned, industry recommendations, and technology improvements. Changes to the Supplements will similarly incorporate lessons-learned as well as definitions of additional services such as Quality of Service monitoring and Fault Management.

Feedback. An open architecture framework is greatly improved through active feedback and recommended changes from a wide audience of potential users. The JTRS JPO solicits and encourages feedback to this document and provides a form available from http://www.jtrs.saalt.army.mil/docs/documents/sca.html. Send the completed form to jtrs.sca@saalt.army.mil. Recommended additions to the SCA must be unencumbered by copyright restrictions or intellectual property rights. Changes to the SCA are controlled by a JTRS JPO-chaired Configuration Control Board (CCB).

1 INTRODUCTION

The Software Communications Architecture (SCA) specification establishes an implementation-independent framework with baseline requirements for the development of Joint Tactical Radio System (JTRS) software configurable radios. These requirements are comprised of interface specifications, application program interfaces (APIs), behavioral specifications, and rules. The goal of this specification is to ensure the portability and configurability of the software and hardware and to ensure interoperability of products developed using the SCA.

Companion documents to this specification are Supplements to the SCA and the SCA Support and Rationale Document (SRD). The Supplements provide specific service and application interface requirements (for Security, networking, other services). The SRD provides the rationale for the SCA and examples to illustrate the implementation of the architecture for differing domains/platforms and selected waveforms.

1.1 SCOPE.

This document provides a complete definition of the SCA. It is an Architecture Framework in that it is precise in areas where reusability is effected and it is general in other areas so that unique requirements of implementations determine the specific application of the architecture. The SCA defines the hardware and software at different levels of detail to allow the broadest reusability and portability of components.

For hardware, the physical and environmental differences across domains are so diverse that physical commonality cannot be achieved for all implementations. However, by using an Object-Oriented (OO) description for the hardware, represented as hardware classes, all potential system implementations are included within a single framework. That framework has attributes (i.e., behavior and interfaces) that are applicable across those different implementations.

The architecture for software makes extensive use of object modeling and its definition is primarily in the Core Framework (CF), an integral part of a system's Operating Environment (OE). Constraints on the software development, imposed by the architecture, are on the interfaces and the structure of the software and not on the implementation of the functions that are performed. In this way, innovative designs can be put forward with appropriate protection of the developer’s intellectual property and still reap the benefits of wide reuse in other implementations of the architecture. The SCA permits either hardware or software to be used in implementing a required function. The approach taken also permits legacy solutions to be incorporated, where appropriate, by encapsulation techniques to provide a “one-sided” standard interface into architecture interfaces.

This architecture specifies rules that further constrain implementations to adhere to open system standards. Specific implementation requirements may augment the rule-set to increase reusability within and across domains.

Figure 1-1 illustrates the concept of the SCA and its implementation down to specific platforms. The hardware definition stays at a framework level with rules providing implementation guidance down into domains and platforms. The software definition can be applied directly down to implementation because of its general independence from hardware implementation. There are special cases where size, weight, and power requirements limit the direct application of software objects. However, even in these cases, reusability of designs, captured in software and firmware modeling and simulation tools, reduces the cost of implementation and the development time.

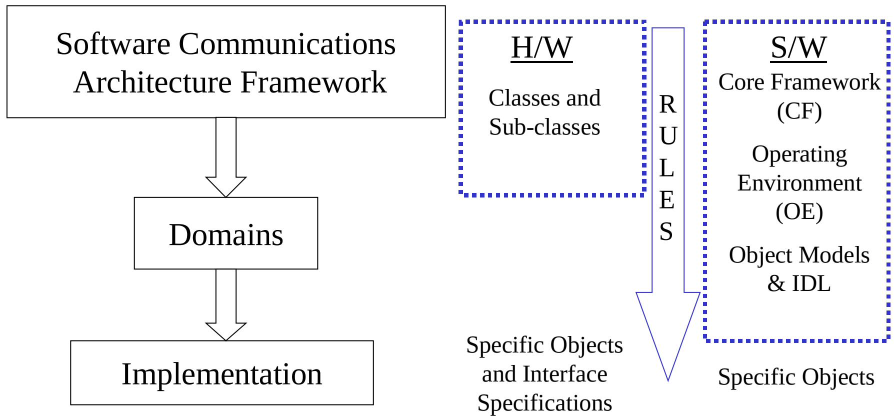

Figure 1-1. The Architecture Framework and its Relationship to Implementation

1.2 COMPLIANCE.

The interfaces, behavior, and rules that define compliance with the SCA are identified in, and are an integral part of this specification. These elements are selected to maximize portability, interoperability, and configurability of the software and hardware while allowing a procurer the flexibility to address domain requirements and restrictions. If any requirements stated in this specification are in conflict with existing standards/specifications, this specification takes precedence.

1.2.1 Joint Technical Architecture Compliance.

The Joint Technical Architecture (JTA) mandates the minimum set of standards and guidelines for all DoD Command, Control, Communications, Computers, and Intelligence (C^4I) systems acquisition. A foremost objective of the JTA is to improve and facilitate the ability of systems to support joint and combined operations in an overall investment strategy. The SCA Operating Environment fully complies with the JTA and provides a JTA-compliant framework for waveforms and other applications.

1.3 DOCUMENT CONVENTIONS, TERMINOLOGY, AND DEFINITIONS.

1.3.1 Conventions and Terminology.

1.3.1.1 Unified Modeling Language.

The Unified Modeling Language (UML), defined by the Object Management Group (OMG), is used to graphically represent SCA interfaces, scenarios, use cases, and collaboration diagrams.

1.3.1.2 Interface Definition Language.

Interface Definition Language (IDL), also defined by the OMG, is used to define the SCA interfaces. IDL is programming language independent and can be compiled into programming languages such as C++, Ada, and Java.

1.3.1.3 eXtensible Markup Language.

eXtensible Markup Language (XML) is used in a Domain Profile to identify the capabilities, properties, inter-dependencies, and location of the hardware devices and software components that make up an SCA-compliant system

1.3.1.4 Color Coding.

Color-coding is used to differentiate between architecture elements and applications in diagrams as shown in Figure 1-2.

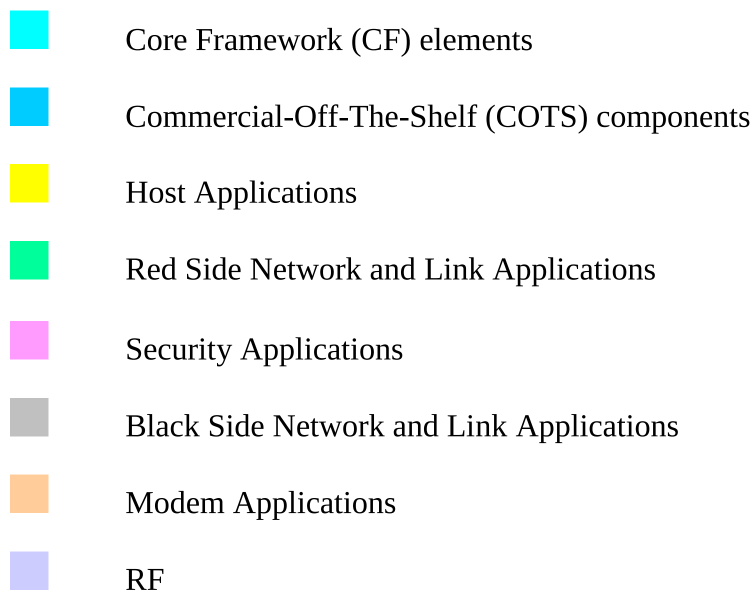

Figure 1-2. Color Coding Used in Document Figures

1.3.1.5 Requirements Language.

Interfaces, behavior, and rules that are imposed by this specification appear in sections 3 through 5 and are indicated by the word "shall". Editorial notes are contained within brackets and are italicized (*{example}*).

1.3.1.6 CF Interface and Operation Identification.

CF interfaces and their operations are presented in italicized text. Core Framework Base Types (3.1.3.5) are prefixed with “CF” when used in textual descriptions (e.g. "each item value is a CF Properties type").

<u>1.3.2 Definitions.</u>

Definitions are included in Appendix A.

1.4 DOCUMENT CONTENT.

This document provides an overview of the SCA in section 2, followed by the Software, Hardware, and Security architecture requirements in sections 3 – 5. Section 6 addresses requirements not contained in those functional categories. Evaluation criteria for product compliance to this specification are addressed in section 7.

Appendices include a glossary, a complete listing of CF IDL, and details of architecture requirements introduced in the main document.

1.5 APPLICABLE DOCUMENTS.

The following documents are applicable to the SCA either by direct reference or as foundation for the architecture definition.

1.5.1 Government Documents.

Joint Technical Architecture, Version 2.0, 26 May 1998.

Operational Requirements Document (ORD) for Joint Tactical Radio (JTR), 23 March 1998.

1.5.2 Commercial Documents.

C Standard: Programming languages – C, ISO/IEC 9899:1990.

DCE UUID standard (OSF Distributed Computing Environment, DCE 1.1 Remote Procedure Call).

"Design Patterns : Elements of Reusable Object-Oriented Software" (Addison-Wesley Professional Computing) Gamma, Helm, Johnson, and Vlissides.

IEEE 802.1 [Medium Access Control (MAC) addresses] IEEE Standards for Local and Metropolitan Area Networks: LAN/MAN Bridging & Management.

ISO/IEC 10731 Conventions for the Definition of OSI Services, Annex D Alternative and Additional Time Sequence Diagrams for Two-party Communications.

minimumCORBA: OMG Document orbos/98-05-13, May 19, 1998.

POSIX 1003.13: Standardized Application Environment Profile - POSIX® Realtime Application Support (AEP), IEEE Std 1003.13-1998.

UML: OMG (Object Management Group) Unified Modeling Language Specification, Version 1.3, March 2000.

XML: W3C (World Wide Web Consortium) Recommendation: Extensible Markup Language (XML) 1.0, Feb 1998.

® POSIX is a registered trademark of the Institute of Electrical and Electronics Engineers, Inc.

2 OVERVIEW

This Section presents an overview of the SCA. Emphasis is on identifying the components of the architecture and the manner in which these components interact. Technical details and requirements of the architecture are contained in Sections 3 - 5.

2.1 ARCHITECTURE DEFINITION METHODOLOGY.

The architecture has been developed using an object-oriented approach wherein the process can be continued beyond the framework definition to product development. UML is used to graphically represent interfaces while IDL is used to define them; both have been generated using standard software development tools, allowing product development to continue directly from the architecture definition.

2.2 ARCHITECTURE OVERVIEW.

2.2.1 Overview - Software Architecture.

The structure of the software architecture is shown in figure 2-1. The key benefits of the software architecture are that it:

1. Maximizes the use of commercial protocols and products,
2. Isolates both core and non-core applications from the underlying hardware through multiple layers of open, commercial software infrastructure, and
3. Provides for a distributed processing environment through the use of the Common Object Request Broker Architecture (CORBA) to provide software application portability, reusability, and scalability.

The software architecture defines an Operating Environment (OE) with the combined set of CF services and infrastructure software (including board support packages, operating system and services, and CORBA Middleware services) integrated in an SCA implementation. The software partitions that illustrate applications are typical of how waveforms might be implemented using the SCA.

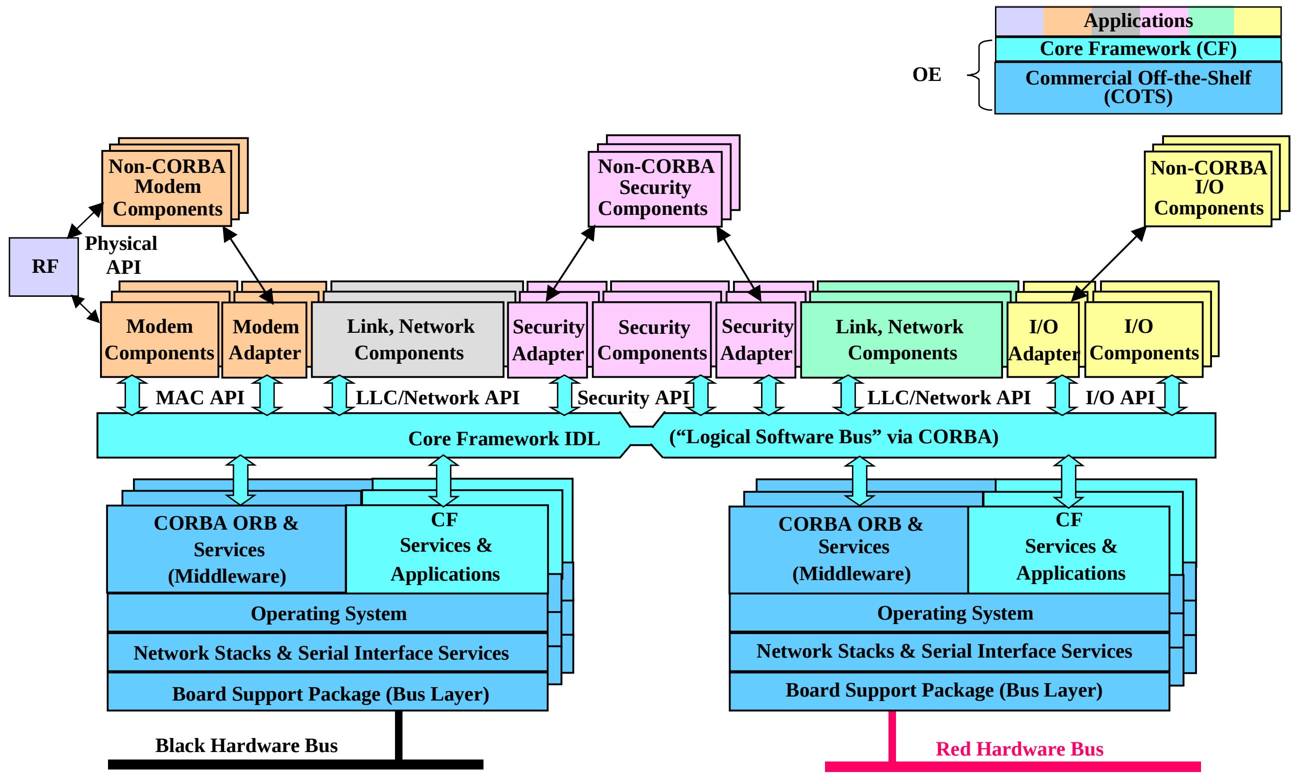

Figure 2-1. Software Structure

2.2.1.1 Bus Layer (Board Support Package).

The Software Architecture is capable of operating on commercial bus architectures. The OE supports reliable transport mechanisms, which may include error checking and correction at the bus support level. Possible buses include VME, PCI, CompactPCI, Firewire (IEEE-1394), and Ethernet. The OE does not preclude the use of different bus architectures on the Red and Black subsystems.

2.2.1.2 Network & Serial Interface Services.

The Software Architecture relies on commercial components to support multiple unique serial and network interfaces. Possible serial and network physical interfaces include RS-232, RS-422, RS-423, RS-485, Ethernet, and 802.x. To support these interfaces, various low-level network protocols may be used. They include PPP, SLIP, LAPx, and others. Elements of waveform networking functionality may also exist at the Operating System layer. An example of this would be a commercial IP stack that performs routing between waveforms.

2.2.1.3 Operating System Layer.

The Software Architecture includes real-time embedded operating system functions to provide multi-threaded support for applications (including CF applications). The architecture requires a standard operating system interface for operating system services in order to facilitate portability of applications.

Portable Operating System Interface (POSIX) is an accepted industry standard. POSIX and its real-time extensions are compatible with the requirements to support the OMG CORBA specification. Complete POSIX compliance encompasses more features than are necessary to control a typical implementation. Therefore, this specification defines a minimal POSIX profile to meet SCA requirements. The SCA POSIX profile is based upon the Real-time Controller System Profile (PSE52) as defined in POSIX 1003.13.

2.2.1.4 Core Framework.

The CF is the essential ("core") set of open application-layer interfaces and services to provide an abstraction of the underlying software and hardware layers for software application designers. Section 3 presents the complete definition of all services and interfaces of the CF. The CF consists of:

1. Base Application Interfaces (*Port, LifeCycle*, *TestableObject*, *PropertySet*, *PortSupplier*, *ResourceFactory*, and *Resource*) that can be used by all software applications,
2. Framework Control Interfaces (*Application*, *ApplicationFactory*, *DomainManager*, *Device*, *LoadableDevice*, *ExecutableDevice*, *AggregateDevice* and *DeviceManager*) that provide control of the system,
3. Framework Services Interfaces that support both core and non-core applications *(File*, *FileSystem*, *FileManager*, and *Timer*), and
4. A Domain Profile that describes the properties of hardware devices (Device Profile) and software components (Software Profile) in the system.

The Domain Profile supports the combination of resources to create applications. Device Profile and Software Profile files utilize an XML vocabulary to describe specific characteristics of either software or device components with regard to their interfaces, functional capabilities, logical location, inter-dependencies, and other pertinent parameters.

2.2.1.5 CORBA Middleware.

CORBA is used in the CF as the message passing technique for the distributed processing environment. CORBA is a cross-platform framework that can be used to standardize client/server operations when using distributed processing. Distributed processing is a fundamental aspect of the system architecture and CORBA is a widely used "Middleware" service for providing distributed processing.

All CF interfaces are defined in IDL. The CORBA protocol provides message marshalling to handle the bit packing and handshaking required for delivering the message. The SCA IDL defines operations and attributes that serve as a contract between components.

2.2.1.6 Application Layer.

Applications perform user communication functions that include modem-level digital signal processing, link-level protocol processing, network-level protocol processing, internetwork routing, external input/output (I/O) access, security, and embedded utilities. Applications are required to use the CF interfaces and services. Applications' direct access to the Operating System (OS) is limited to the services specified in the SCA POSIX Profile. Networking functionality that may be implemented below the application layer, such as a commercial IP network layer, is not limited to the SCA POSIX Profile since it exists in the OS kernel space.

2.2.1.6.1 Applications.

Applications consist of one or more *Resources*. The *Resource* interface provides a common API for the control and configuration of a software component. The application developers can extend these definitions by creating specialized *Resource* interfaces for the application. At a minimum, the extension inherits the *Resource* interface. Examples of *Resource* extensions are: *LinkResource*, *NetworkResource*, and *UtilityResource*.

Devices are types of *Resources* used by applications as software proxies for actual hardware devices. *ModemDevice, I/ODevice,* and *SecurityDevice* are examples that implement the *Device* interfaces.

ModemDevice, *LinkResource*, *SecurityDevice*, *I/ODevice*, and *NetworkResource* are Core Framework interface extensions that implement APIs for waveform and networking applications.

The design of a *Resource's* internal functionality is not dictated by the Software Architecture. This is left to the application developer. Core applications, which are a part of the CF, support the non-core applications by providing the necessary function of control as well as standard interface definitions. The interfaces by which a *Resource* is controlled and communicates with other *Resources* are defined in section 3.

2.2.1.6.2 Adapters.

Adapters are *Resources* or *Devices* used to support the use of non-CORBA capable elements in the domain. Adapters are used in an implementation to provide the translation between non-CORBA-capable components or devices and CORBA-capable *Resources*. The Adapter concept is based on the industry-accepted Adapter design pattern[1]. Since an Adapter implements the CF CORBA interfaces known to other CORBA-capable *Resources,* the translation service is transparent to the CORBA-capable *Resources*. Adapters become particularly useful to support non-CORBA-capable Modem, Security, and Host processing elements. Figure 2-2 depicts an example of message reception flow through the system with and without the use of Adapters. Modem, Security, and Host Adapters implement the interfaces marked by the circled letters M, S, and H respectively. Notice that the Waveform Link and Network *Resources* are unaffected by the inclusion or exclusion of the Adapters. The interface to these *Resources* remains the same in either case.

[1] "Design Patterns : Elements of Reusable Object-Oriented Software" (Addison-Wesley Professional Computing) Gamma, Helm, Johnson, and Vlissides, pg. 139

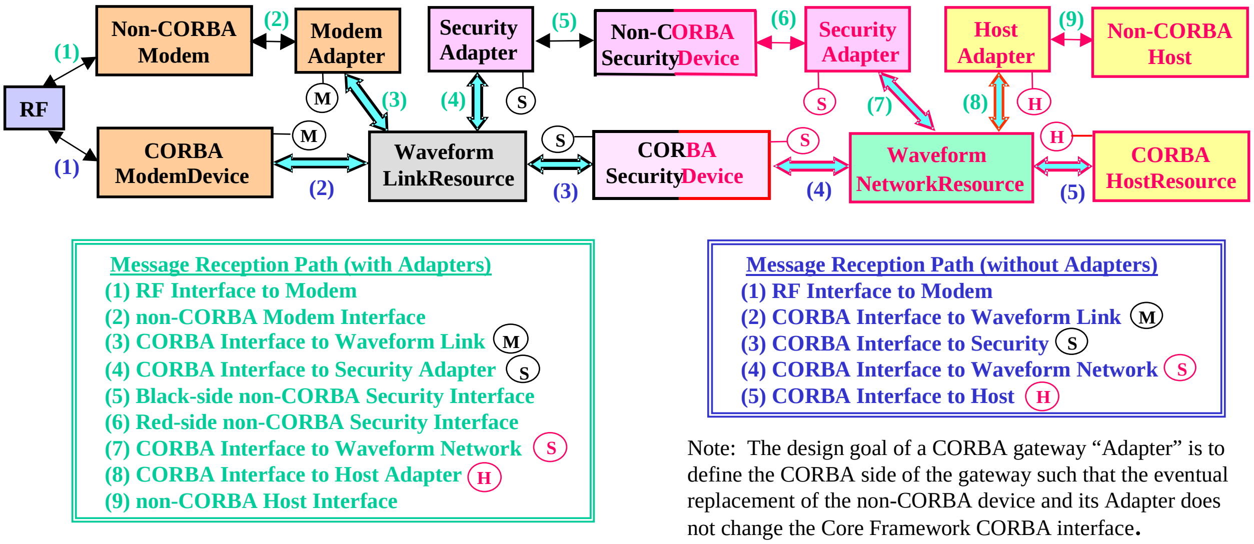

Figure 2-2. Example Message Flows with and without Adapters

2.2.1.7 Software Radio Functional Concepts.

2.2.1.7.1 Software Reference Model.

The software reference model depicted in Figure 2-3 is based upon the Programmable Modular Communication System (PMCS) Reference Model. This model forms a basis for the SCA by:

1. Introducing the various functional roles performed by software entities without dictating a structural model of these elements, and
2. Introducing the control and traffic data interfaces between the functional software entities.

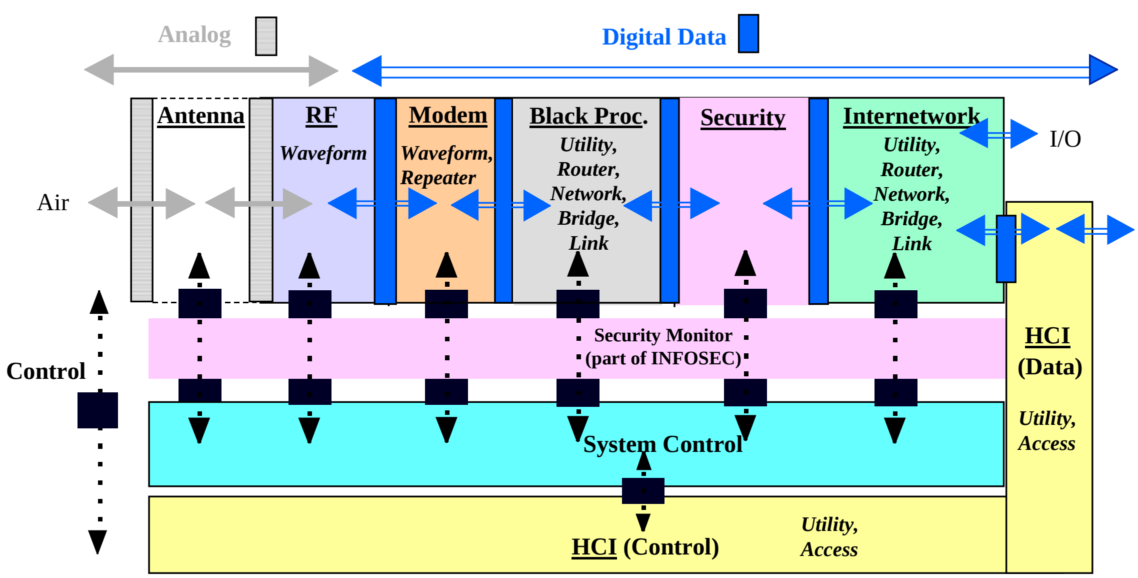

Figure 2-3. Software Reference Model

The Reference Model identifies relevant functionality but does not dictate the architecture. The SCA realizes the Software Reference Model by defining a standard unit of functionality called a *Resource*. All applications are comprised of *Resources* and using *Devices*. Specific resources and devices can be identified corresponding to the functional entities of the Software Reference Model:

ModemDevice:	addresses Antenna, RF, and Modem entities,
LinkResource:	addresses Black Processing entity,
SecurityDevice:	addresses Security entity,
NetworkResource:	addresses Internetworking entity,
I/ODevice:	addresses external interfaces such as serial, ethernet, and audio
UtilityResource:	addresses non-Waveform functionality.

System control entity functionality is addressed by the core framework applications: *Application, ApplicationFactory*, *DomainManager*, *Device*, *LoadableDevice, ExecutableDevice*, *AggregateDevice*, and *DeviceManager*. Control functionality may also be localized in individual resources.

Figure 2-4 shows examples of implementation classes for Resources. The operations and attributes provided by *LifeCycle*, *TestableObject*, *PortSupplier*, and *PropertySet* establish a common approach for interacting with any resource in a SCA environment. *Port* can be used for pushing or pulling messages between *Resources* and *Devices*. A *Resource* may consist of zero or more input and output message ports. The figure also shows examples of more specialized resources and devices that result in specific functionality for each of six example types. Clarification of the functionality associated with each of those is provided in the following subsections. Examples of *Devices* in the following sections and figures can be examples of *Device, LoadableDevice, ExecutableDevice, AggregateDevice,* and *DeviceManager*.

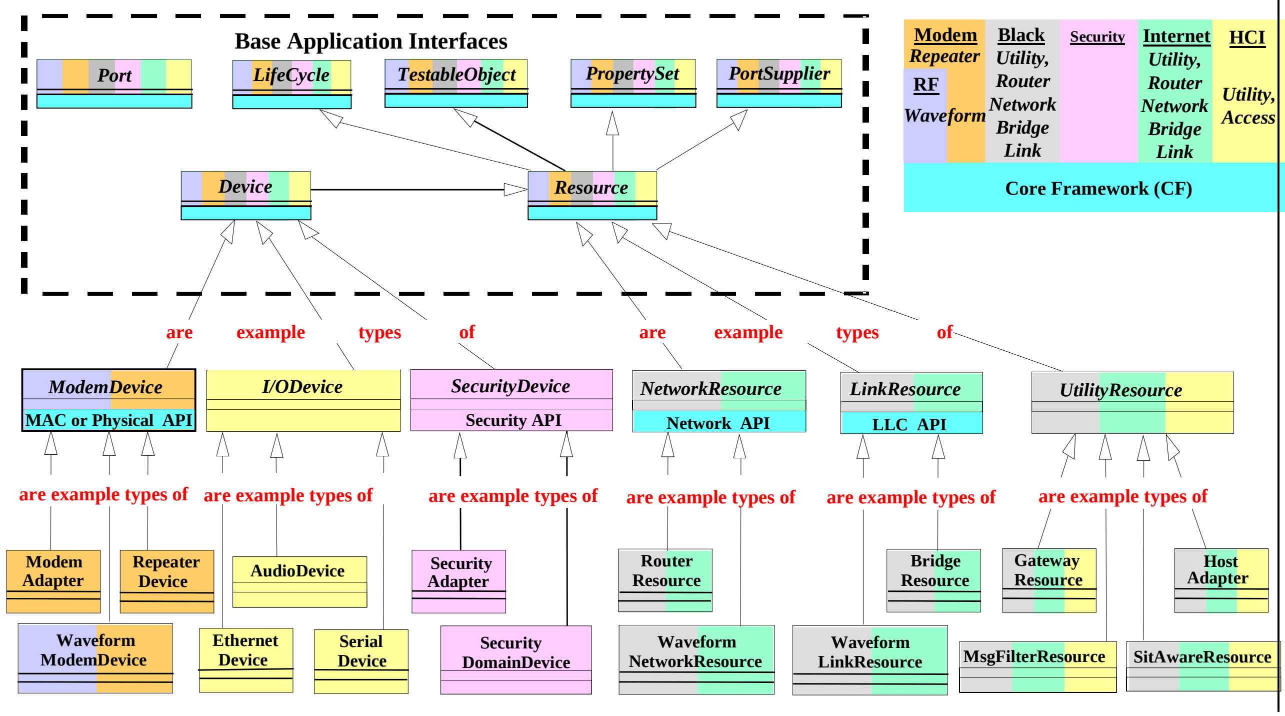

Figure 2-4. Conceptual Model of Resources

2.2.1.7.2 *ModemDevice* Functionality.

The *ModemDevice* provides a standard for the control and interface of a modem, which encapsulates diverse implementations of smart antenna, RF, and modem functions. The base application interfaces are extended to modem devices through a Physical, Medium Access Control (MAC), or Logical Link Control (LLC) API (see the API Supplement to the SCA), which provides a standard interface for control and communication with modem operations from a higher (e.g., link layer to a MAC) resource. The functions, performed by the *ModemDevices*, will vary depending on waveform requirements as well as hardware/software allocation and are not dictated by the CF. Typical RF and modem functions are depicted in Figure 2-5.

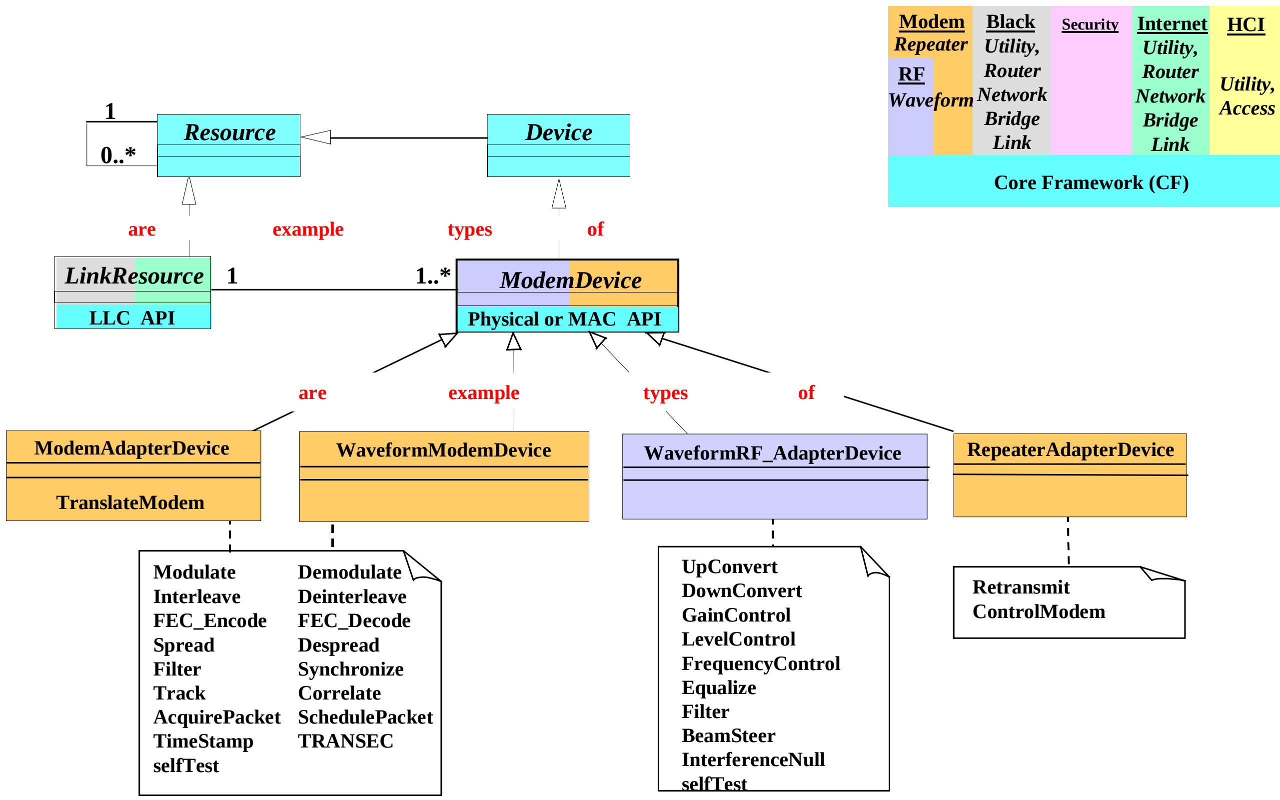

Figure 2-5. Example of Modem Resources

2.2.1.7.3 *NetworkResource* and *LinkResource* Functionality.

An example of networking resources is shown in figure 2-6. The CF base application interfacesare extended to link layer and network layer resources through APIs (see section 2.2.2.2), provided to enable information transfer and support of specific service characteristics for networking applications. Examples are the Link-LLC API and Network-MAC API, which provide standard interfaces for control and communication between network, link, and transport layer resources.

The functions performed by the waveform networking and internetworking resources (examples shown in note boxes in figure 2-6) will vary depending on waveform requirements as well as networking requirements and are not dictated by the CF. *Resources* that provide networking behavior, including repeater, link, bridge, network, router, and gateway operations, are representative and not defined in the SCA.

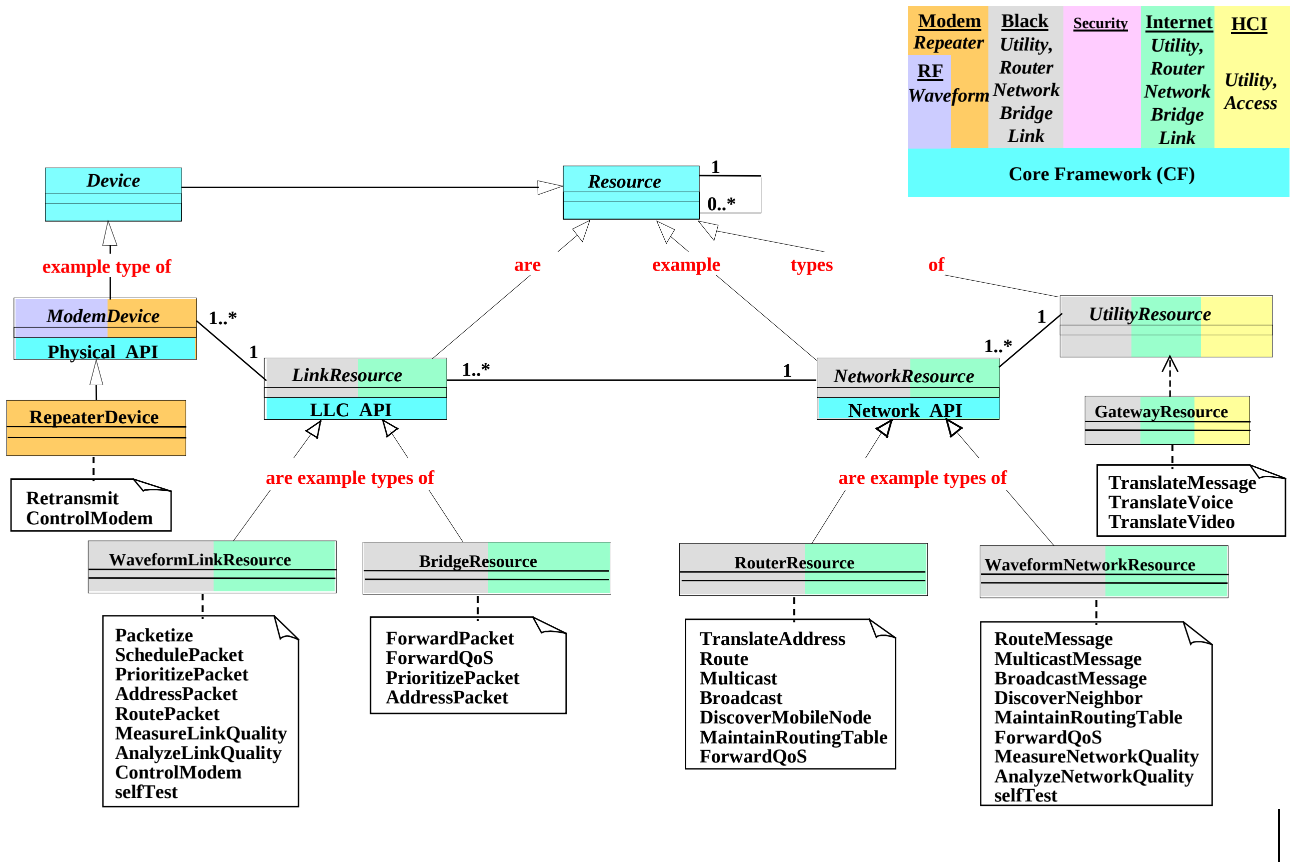

Figure 2-6. Example of Networking Resources

2.2.1.7.4 *I/ODevice* Functionality.

Examples of *I/ODevices* are shown in figure 2-7. An *I/ODevice* provides access to system hardware devices and external physical interfaces. The operations performed by an *I/ODevice* will vary depending on the system hardware assets as well as the physical interfaces to be supported and are not dictated by the CF. Typical I/O operations are depicted within the example subclasses.

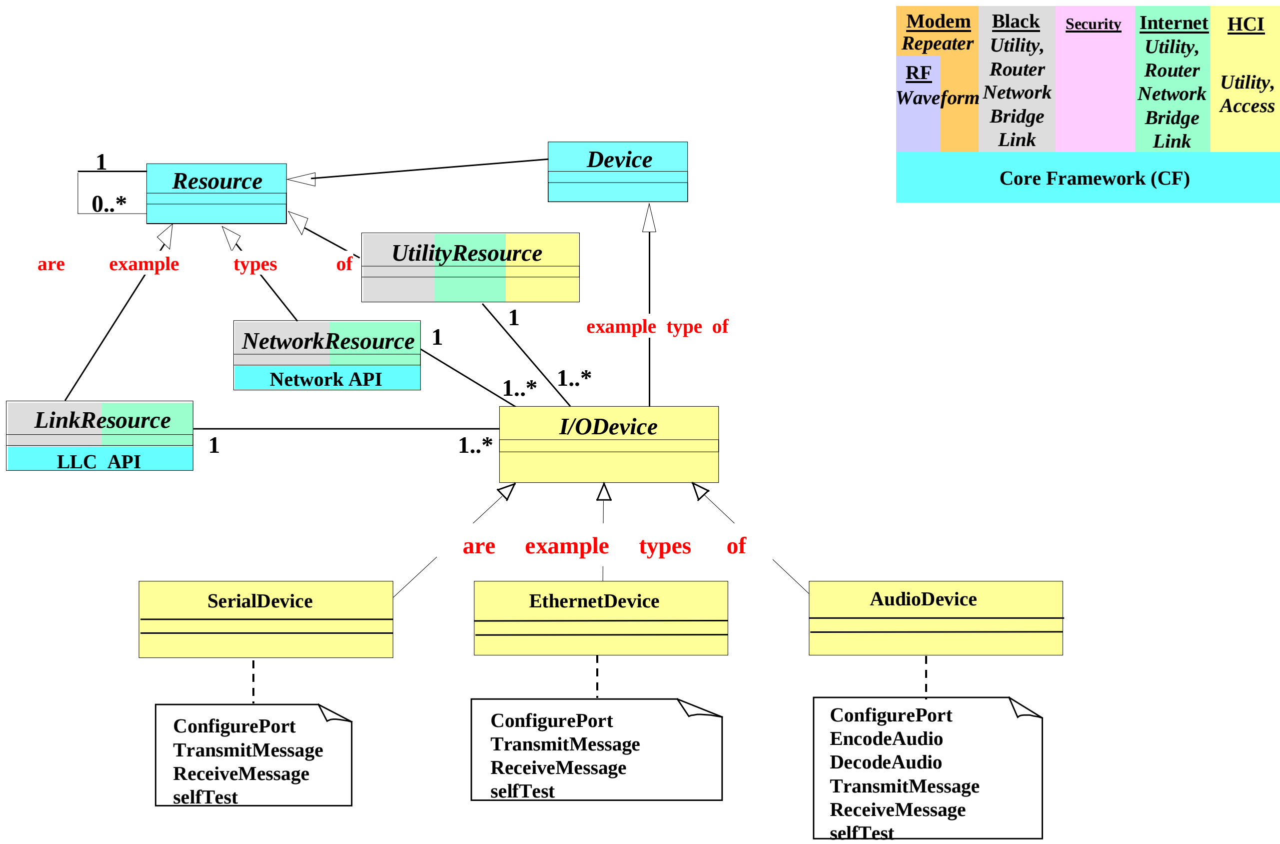

Figure 2-7. Examples of I/O Resources

2.2.1.7.5 *SecurityDevice* Functionality.

Examples of *SecurityDevice* and *SecurityResource* are shown in figure 2-8. Typical security operations are depicted within the example subclasses. *SecurityDevice* subclasses extend security functions to hardware devices within the system while *SecurityResource* subclasses extend security functions to software components. There can be a wide variation of security solutions both in hardware and software. Transmission security (TRANSEC) and communications security (COMSEC) requirements also vary between waveforms. The location of the security boundary with respect to networking requirements also varies between waveforms. The CF base application interfaces are extended to *SecurityResources* through Security APIs, which provide standard interfaces for control and communication between security devices and resources and application waveforms.

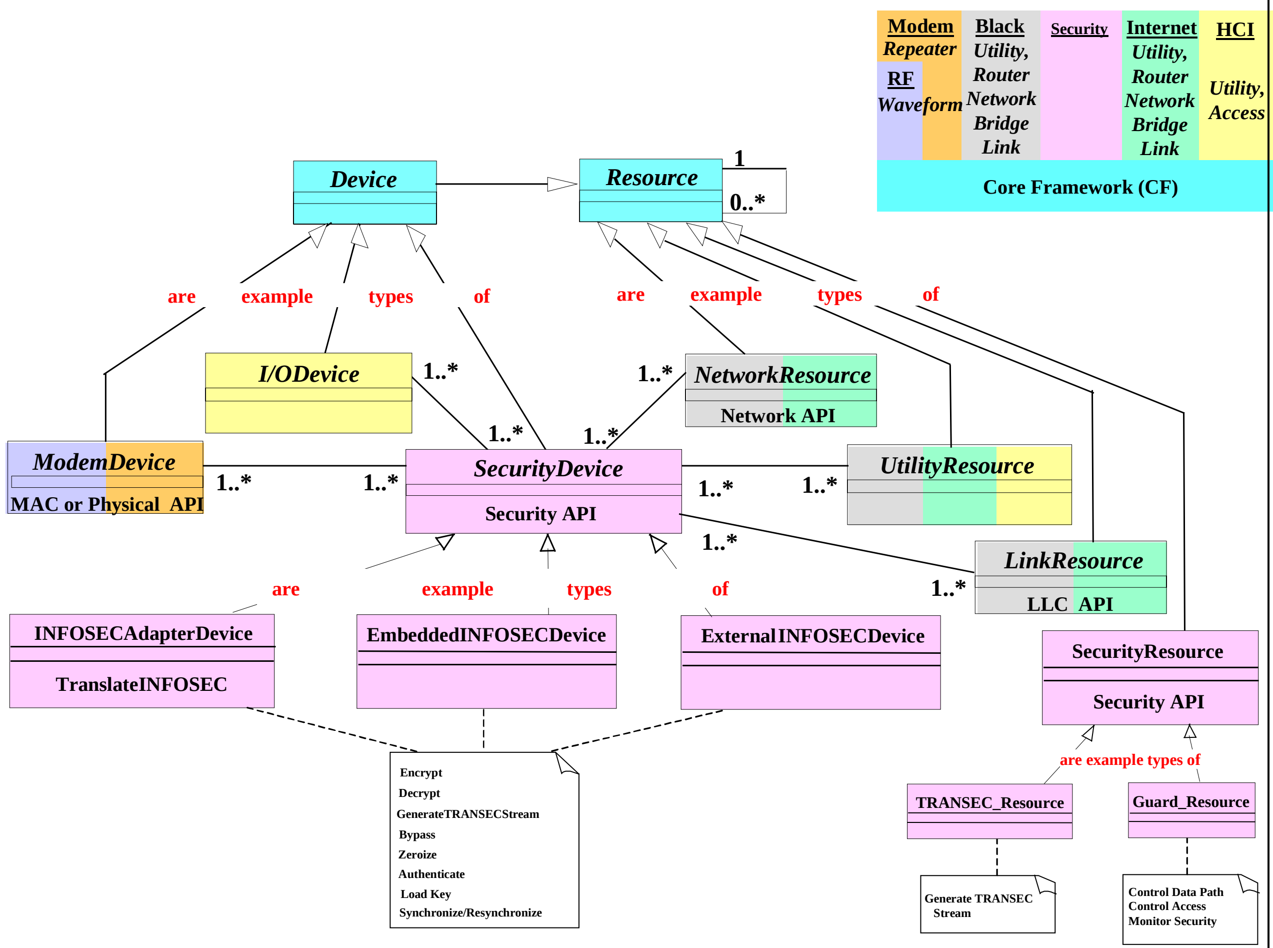

Figure 2-8. Examples of Security Devices and Resources

2.2.1.7.6 *UtilityResource* Functionality.

An example of *UtilityResource* is shown in Figure 2-9. The operations performed by the utility resources will vary depending on the embedded applications to be supported as well as host interface protocol requirements and are not dictated by the CF. Typical utility operations are depicted within the example subclasses. Ultimately, the *UtilityResource* encompasses any non-waveform application that could execute in an SCA-compliant system.

2.2.1.8 System Control.

The SCA provides a specification for interfaces, services, and data formats for the control of resources. Each resource establishes its controllable parameters with the *DomainManager* via a Domain Profile. Applications constrain each resource's parameter values to their own needs. Applications' controllable parameters are also in the Domain Profile.

Use of CORBA and the base application interfaces provides the means to have domain and application control though a common interface. SerialDevice and EthernetDevice (in Figure 2-7) are examples of the external interfaces available to a user. These examples show that system control operations operate with human or machine interfaces either locally or remotely and interact in a manner that facilitates portability.

Non-CORBA user terminals are interfaced through the use of Adapters.

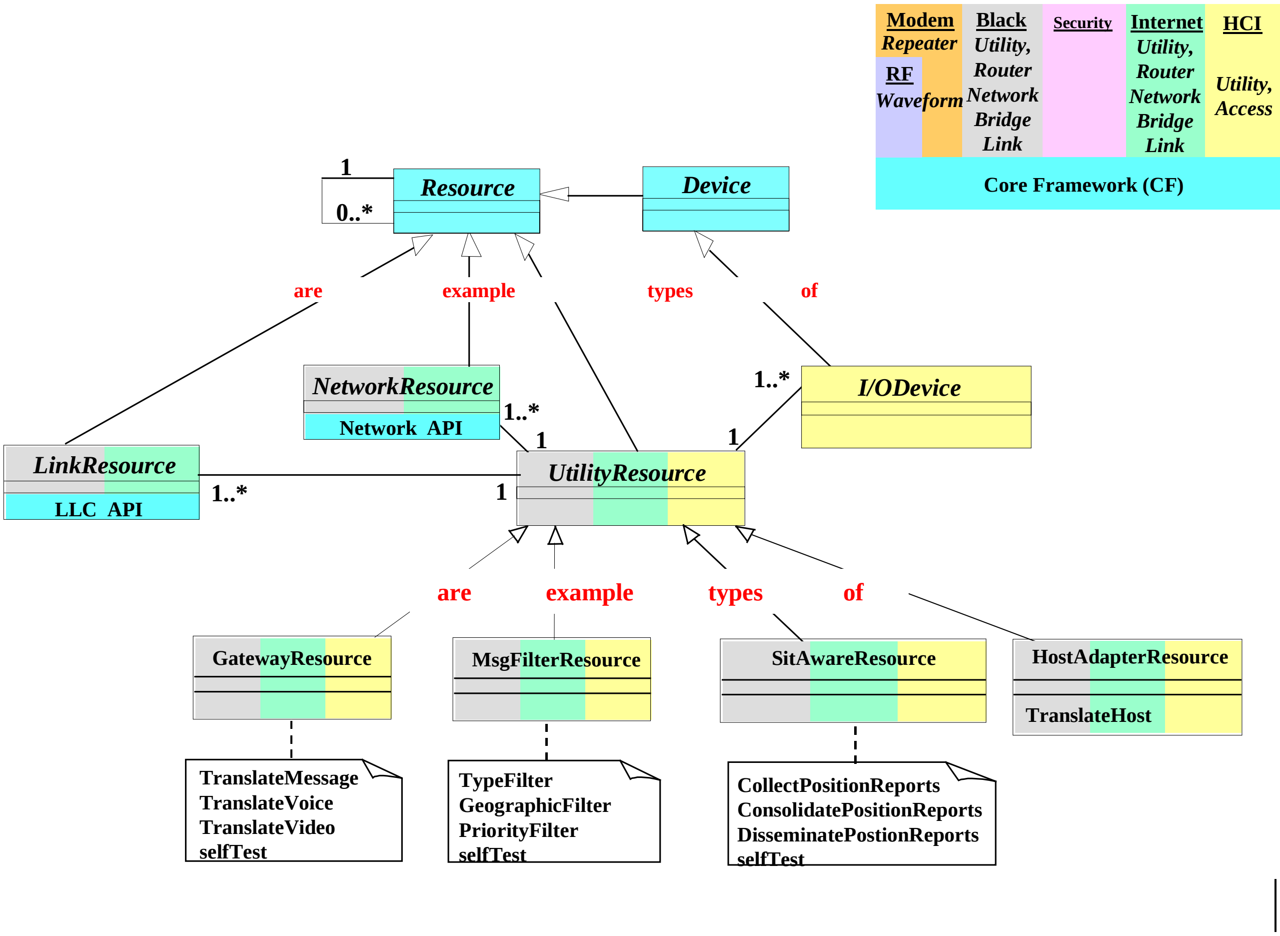

Figure 2-9. Example of Utility Resources

2.2.2 Networking Overview.

SCA-compliant Radio Systems communicate with peer systems through protocols as shown in figure 2-10. The external networking protocols between an SCA-compliant System and its peers are part of waveform applications and are not specified by this architecture specification. However, the interface definitions for the services required to implement the protocols within an SCA-compliant System are specified (in the API Supplement).

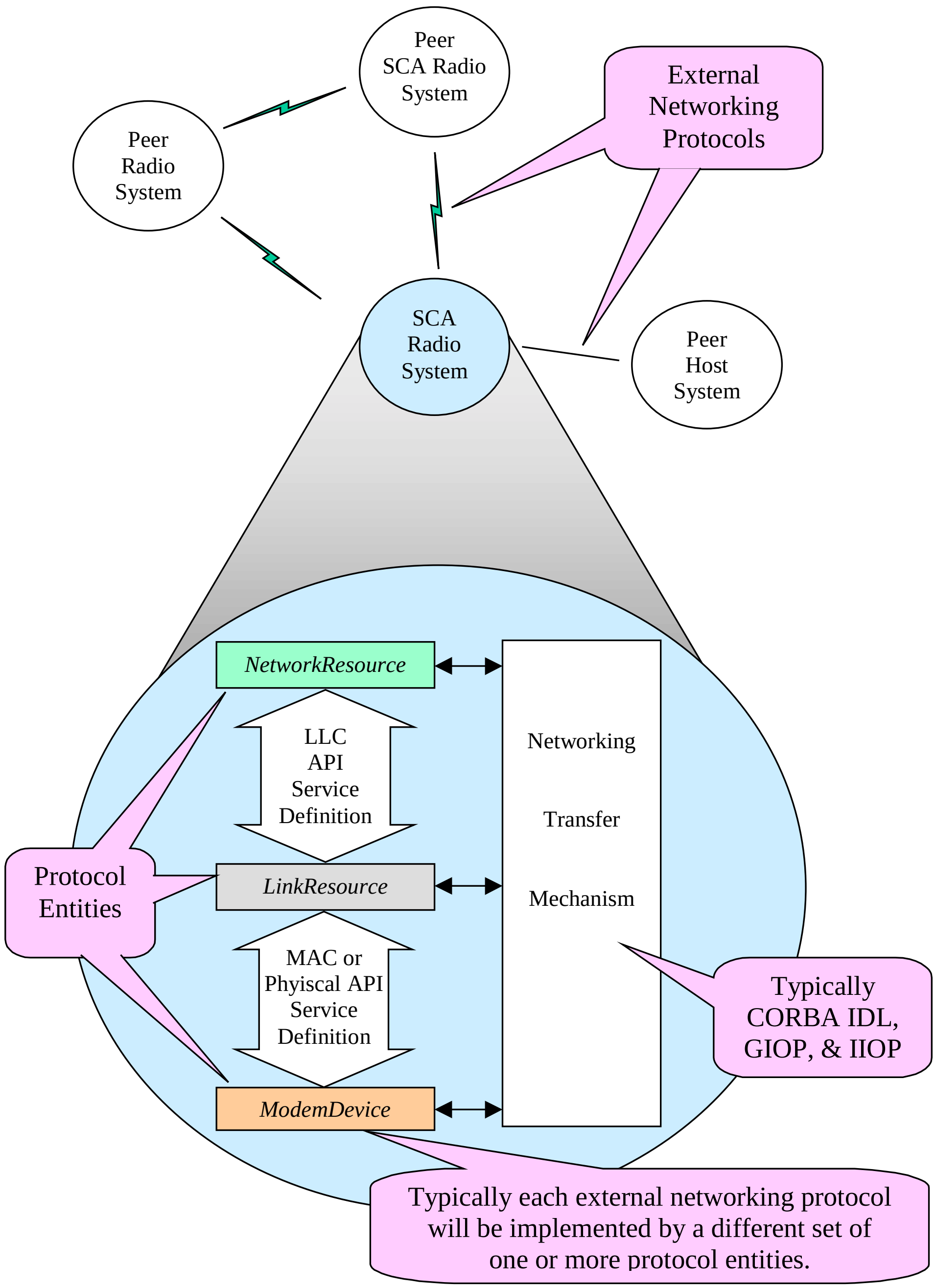

Figure 2-10. External Network Protocols and SCA Support

2.2.2.1 External Networking Protocols.

External networking protocols define the communications between an SCA-compliant Radio System and its peer systems. These external networking protocols can run over wireless or wireline physical media. Example protocols include Single Channel Ground/Airborne Radio System (SINCGARS), Ethernet, HF Automatic Link Establishment (ALE), IEEE 802.11, IS-95A, IP, and future networking protocols.

Through the external networking protocols, implemented by applications in an SCA-compliant radio system and its peer systems, a network of nodes is formed interconnected by repeaters, bridges, routers, and/or gateways. As shown in figure 2-11, external networking protocols will typically interconnect at different layers using:

1. Physical layer interconnections with a repeater function,
2. Link layer interconnections with a bridge function,
3. Network layer interconnections with standard network routing, and/or
4. Upper layer interconnections with application gateways.

The different categories of interoperability are outlined below based upon the OSI Model. There may be multiple levels of interoperability within the same system on a waveform by waveform basis.

A. Physical Layer Interoperability. The external networking protocols provide a compatible physical interface, including the signaling interface, but no higher layer processing. This level of interoperability is adequate for a simple bit by bit bridging or relay operation between two interfaces.

B. Link Layer Interoperability. The external networking protocols provide link layer processing over all physical interfaces. This level of interoperability is adequate for allowing the radio to be used as transport and for allowing the radio to use another network as transport. Intelligent routing or switching decisions are limited to local layer 2 routing.

C. Network Layer Interoperability. The external networking protocols provide network layer address processing interoperability. The radio and the networks being inter-operated are sub-networks of the same Inter-network. At this level, intelligent switching and routing decisions can be made end-to-end.

D. Host Level Interoperability (Layers 4 – 7). Embedded applications can exchange information with hosts attached to the network. An example of this is a handheld radio that contains an embedded Situation Awareness (SA) application exchanging SA updates with a vehicular platform in an external sub-network. In this example, the radio provides message payload translations to allow two otherwise incompatible hosts to communicate.

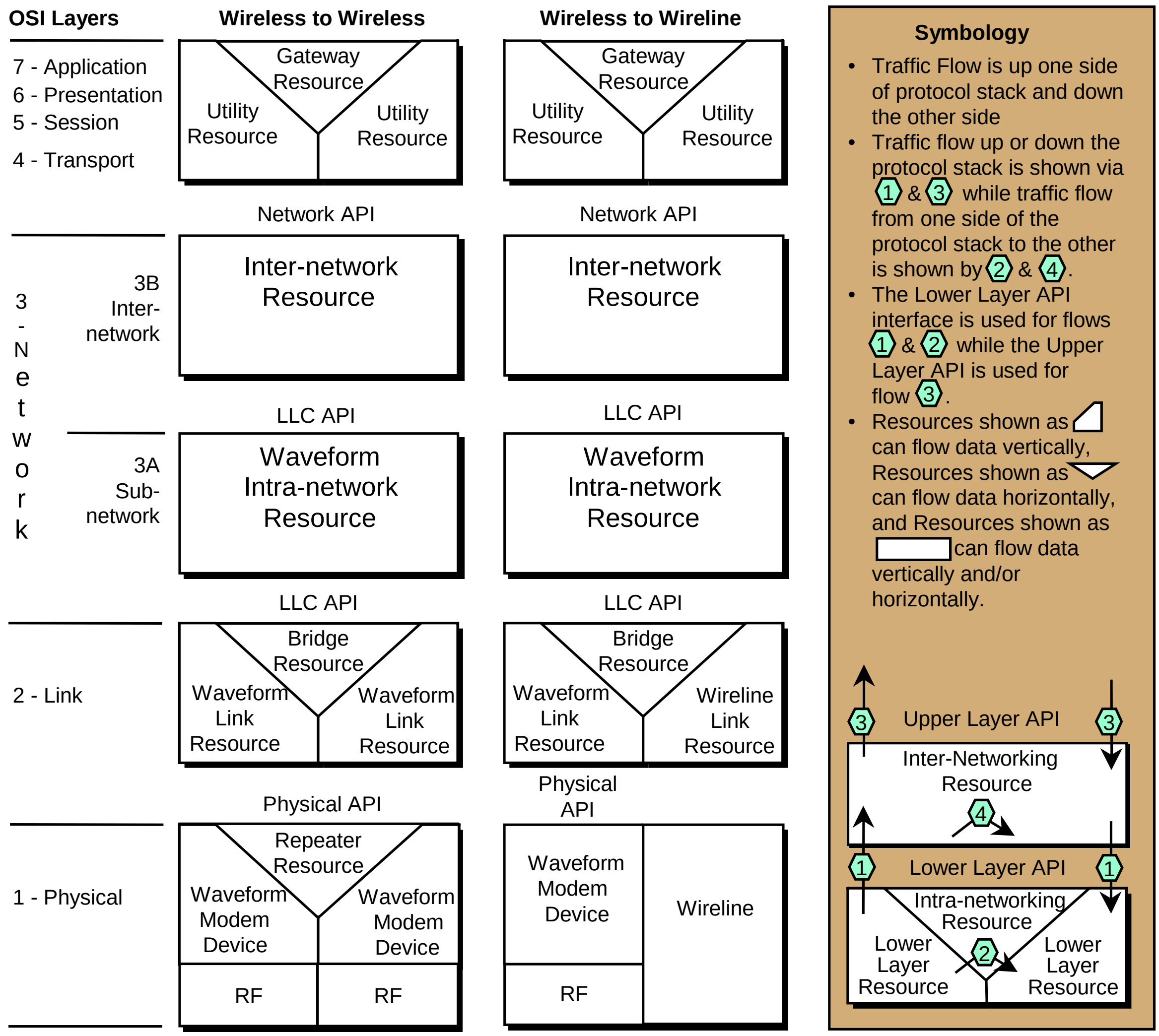

Figure 2-11. SCA-Supported Networking Mapped to OSI Network Model

2.2.2.2 SCA Support for External Networking Protocols.

Figure 2-10 shows that within an SCA-compliant Radio System, application protocol entities are used to implement the external networking protocols. These protocol entities are networking applications[2]. Entity types that support external networking protocols include *ModemDevice*,

---

[2] External networking protocol entities can reside within an application or within the kernel space of operating systems. These external networking protocol applications are not necessarily the same as OSI layer 7 applications. (When an application uses protocol entities within the OS kernel space, and that kernel space is also used for internal system CORBA transport protocol,

LinkResource, *NetworkResource*, *SecurityDevice*, *I/ODevice,* and *UtilityResource.* Typically, each waveform or wireline protocol will be implemented by a unique set of one or more protocol entities. A unique set of protocol entities implements the protocol stack specified by a waveform or wireline protocol. A radio system implementing multiple waveform applications may have multiple protocol entities at each protocol layer.

In order to support application portability, standard interfaces are required between application protocol entities. These Networking APIs, support the concept of a service interface between a service provider (usually the lower OSI protocol layer) and a service user (usually the higher OSI protocol layer).

Networking APIs, like other waveform application APIs, are extensions to the CF base application interfaces that are inherited from the CF *Resource* class. APIs can be extended allowing vendors to provide value-added features that distinguish themselves from their competitors.

Two Networking API types are illustrated in this section: an LLC API associated with the *LinkResource* and a Network API associated with the *NetworkResource.* The APIs can be mapped into the OSI Networking Protocol model as shown in figure 2-11. This figure shows two very similar protocol stacks for wireless-to-wireless networking and wireless-to-wireline networking. The difference is that the wireline stack has a WirelineDevice at the physical layer instead of a *ModemDevice*. (Note that the OSI network layer maybe split into multiple network resources as shown in figure 2-11. In most cases, the layer 3A sub-network has an LLC API to the upper layer 3B inter-network (for example when layer 3B is IP). However, for some network waveform protocols, the layer 3A interface may be the network API).

The SCA defines an API Instance to provide the mechanism for distributing the protocol layers within an SCA-compliant Radio System. An API Instance is a coupling of a Networking API Service Definition and a Transfer Mechanism for a particular waveform implementation. The Service Definition for a waveform details the primitives (operations), the parameters (variables), their representation (structures, types, formats), and its behavior. The transfer mechanism provides the communication between the waveform protocol layer service provider and a service user. CORBA is the preferred transfer mechanism. Because security requirements for a particular implementation may be met using services associated with CORBA, later introduction of a different transfer mechanism requires careful analysis of the security services that can be provided by that transfer mechanism. Figure 2-10 shows the relationship between protocol entities, Service Definitions, and Transfer Mechanisms.

2.2.3 Overview - Hardware Architecture.

Partitioning the hardware into classes places emphasis on the physical elements of the system and how they are composed of functional elements. These classes define common elements sharing physical attributes (characteristics and interfaces) that carry over to implementation for specific domain platforms. The same framework applies to all domains. Appropriate application of the requirements leads to common hardware modules for different platforms. A summary view of the hardware framework is shown in figure 2-12.

---

additional security protection may be required to prevent external network nodes from directly connecting with internal CORBA objects.)

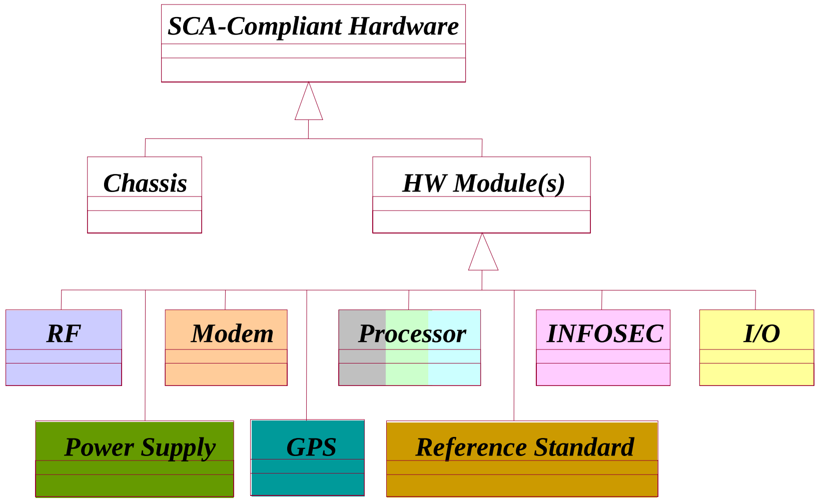

Figure 2-12. Hardware Architecture Framework

The *HWModule(s)* class inherits the system level attributes from the *SCA-CompliantHardware* class. Classes below the *HWModule(s)* class inherit the attributes of that class. The attributes are the parameters that define domain-neutral hardware devices, and the values assigned to the attributes satisfy requirements for a selected implementation. The hardware devices, which are the physical implementation of these classes, will have values for the relevant attributes based on a platform's physical requirements and the procurement performance requirements. Some attributes are used in the creation of waveform applications and provided in a Device Profile, readable by CF applications.

The Chassis Class has unique physical, interface, platform power, and external environment attributes that are not shared with the modules in the chassis. Software Architecture Definition

3 OPERATING ENVIRONMENT.

This section contains the requirements of the operating system, middleware, and the CF interfaces and operations that comprise the OE.

3.1.1 Operating System.

The processing environment and the functions performed in the architecture impose differing constraints on the architecture. An SCA application environment profile (AEP) was defined to support portability of waveforms, scalability of the architecture, and commercial viability. POSIX specifications are used as a basis for this profile. The notional relationship of the OE and applications to the SCA AEP is depicted in figure 3-1. The OS shall provide the functions and options designated as mandatory by the AEP defined in Appendix B. The OS is not limited to providing the functions and options designated as mandatory by the profile. The CORBA Object Request Broker (ORB), the CF Framework Control Interfaces, Framework Services Interfaces, and hardware device drivers are not limited to using the services designated as mandatory by the profile.

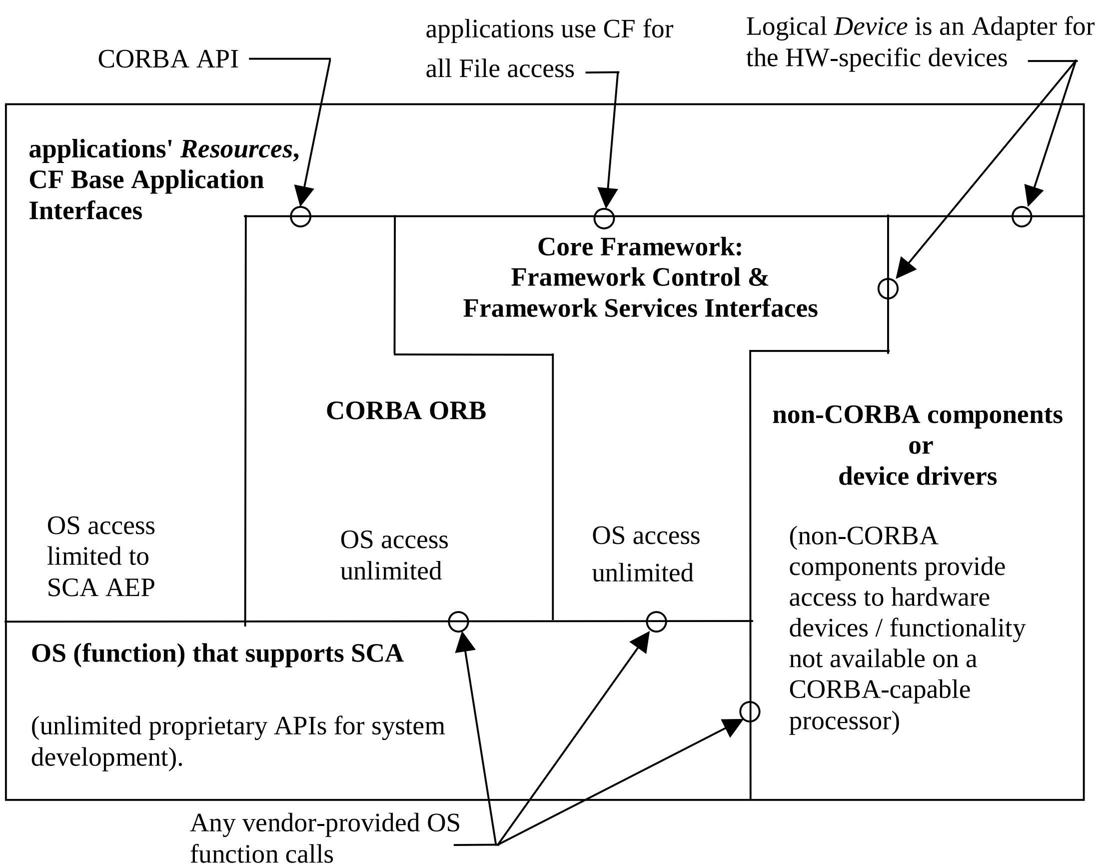

Figure 3-1. Notional Relationship of OE and Application to the SCA AEP

The OS and related file systems shall support a minimum file name length of 40 characters and a minimum combined pathname/filename length of 1024 characters.

Applications are limited to using the OS services that are designated as mandatory for the profile. Applications will perform file access through the CF. (Application requirements are covered in section 3.2.)

3.1.2 Middleware & Services.

3.1.2.1 CORBA.

The OE shall use middleware that, at a minimum, provides the services and capabilities of minimumCORBA as specified by the OMG Document orbos/98-05-13, May 19, 1998.

3.1.2.2 CORBA Extensions.

The following extensions and/or services above and beyond minimumCORBA are allowed.

3.1.2.2.1 Naming Service.

CORBA Naming Service may be used. If a CORBA Naming Service is used, the OE should provide an Interoperable Naming Service as specified by the OMG Document orbos/98-10-11, October 19, 1998.

As an alternative, software components will include stringified Interoperable Object References (IORs) in their Software Profile. If stringified IORs are used, the object reference shall be generated using the persistent LifeSpan policy per the minimumCORBA specification.

3.1.2.3 Log Service.

3.1.2.3.1 Use of Log Service.

This section describes the requirements for components that produce log records. A log producer is a CF component (e.g., *DomainManger*, *Application*, *ApplicationFactory*, *DeviceManager*, *Device*) or an application's CORBA capable component (e.g., *Resource*, *ResourceFactory*) that produces log records. (A component that calls the *writeRecords* operation of the Log interface.)

A standard record type is defined for all log producers to use when writing log records. The log producer may be configured via the *PropertySet* interface to output only specific log levels.

Log producers shall implement a configure property with an ID of "PRODUCER_LOG_LEVEL". The PRODUCER_LOG_LEVEL configure property provides the ability to "filter" the log message output of a log producer. The type of this property shall be a LogLevelSequence. The configure property *LogLevelSequence* contains all log levels that shall be enabled. Only the messages that contain an enabled log level may be sent by a log producer to a Log. Log levels which are not in the LogLevelSequence shall be disabled.

Log producers shall implement a property with an ID of "PRODUCER_LOG_ID". The PRODUCER_LOG_ID configure property provides an instance unique identifier for the log producer. The log producer shall use the string value of the PRODUCER_LOG_ID property in the ProducerID field of the ProducerLogRecord.

Log producers shall operate normally in the case where the connections to a Log are nil or an invalid reference.

Log producers shall output only those log records that correspond to enabled LogLevelType values.

3.1.2.3.2 LogServiceModule.

The LogService module contains the Log servant interface and the types necessary for a log producer to generate standard SCA log records. This module also defines the types necessary to control the logging output of a log producer. Components that produce logs are required to implement configure properties that allow the component to be configured as to what log records it will output.

Implementation of the LogService is optional (i.e. a log service is not a required component of a JTRS installation). CF components that are required to write log records are also required to account for the absence of a log service and otherwise operate normally.

3.1.2.3.2.1 Types.

3.1.2.3.2.1.1 LogLevelType.

Type LogLevelType is an enumeration that is utilized to identify log levels.

```
enum LogLevelType {SECURITY_ALARM
            , FAILURE_ALARM
            , DEGRADED_ALARM
            , EXCEPTION_ERROR
            , FLOW_CONTROL_ERROR
            , RANGE_ERROR
            , USAGE_ERROR
            , ADMINISTRATIVE_EVENT
            , STATISTIC_REPORT
            , PROGRAMMER_DEBUG1
            , PROGRAMMER_DEBUG2
            , PROGRAMMER_DEBUG3
            , PROGRAMMER_DEBUG4
            , PROGRAMMER_DEBUG5
            , PROGRAMMER_DEBUG6
            , PROGRAMMER_DEBUG7
            , PROGRAMMER_DEBUG8
            , PROGRAMMER_DEBUG9
            , PROGRAMMER_DEBUG10
            , PROGRAMMER_DEBUG11
            , PROGRAMMER_DEBUG12
            , PROGRAMMER_DEBUG13
            , PROGRAMMER_DEBUG14
            , PROGRAMMER_DEBUG15
            , PROGRAMMER_DEBUG16
            }
```

3.1.2.3.2.1.2 ProducerLogRecordType.

Log producers format log records as defined in the structure ProducerLogRecordType.

```
struct ProducerLogRecordType {
     unsigned long producerID;
     string producerName;
     LogLevelType level;
```

```
     String logData;
}
```

producerID : This field uniquely identifies the source of a log record. The value is unique within the Domain. The *DomainManager* and *ApplicationFactory* are responsible for assigning this value.

producerName : This field identifies the producer of a log record in textual format. This field is assigned by the log producer, thus is not unique within the Domain (e.g. multiple instances of an application will assign the same name to the ProducerName field.)

level : This field identifies the type of message being logged as defined by the type LogLevelType.

logData : This field contains the informational message being logged.

3.1.2.3.2.1.3 LogLevelSequence.

The LogLevelSequence type is an unbounded sequence of LogLevelTypes. The PRODUCER_LOG_LEVEL configure/query property is of the LogLevelSequence `type.`

```
typedef sequence <LogLevelType> LogLevelSequence;
```

3.1.2.3.3 Log.

3.1.2.3.3.1 Description.

A Log is utilized by CF and CORBA capable application components to store informational messages. These informational messages are referred to as ‘log records’ in this document. The interface provides operations for writing log records to a Log, retrieving log records from a Log, control of a Log, and status of a Log.

3.1.2.3.3.2 UML.

Figure 3-2. Log UML.

3.1.2.3.3.3 Types.

3.1.2.3.3.3.1 InvalidParamException.

The InvalidParamException exception indicates that a provided parameter was invalid.

```
exception InvalidParam {string details};
```

3.1.2.3.3.3.2 LogEmptyException.

The LogEmptyException exception indicates that the log is empty.

```
exception LogEmpty ();
```

3.1.2.3.3.3.3 LogTimeType.

This type provides the time format used when writing log records. The Log implementation is required to produce time-stamps compatible with the POSIX-defined *struct tm*.

```
typedef unsigned long long LogTimeType;
```

3.1.2.3.3.3.4 OperationalStateType.

The enumeration OperationalStateType defines the Log states of operation. When the Log is ENABLED it is fully functional and is available for use by log producer and log consumer clients. A Log that is DISABLED has encountered a runtime problem and is not available for use by log producers or log consumers. The internal error conditions that cause the Log to set the operational state to ENABLED or DISABLED are implementation specific.

```
enum OperationalStateType {DISABLED, ENABLED};
```

3.1.2.3.3.3.5 AdministrativeStateType.

The AdministrativeStateType denotes the active logging state of an operational Log. When set to UNLOCKED the Log will accept records for storage, per its operational parameters. When set to LOCKED the Log will not accept new log records and records can be read or deleted only.

```
enum AdministrativeStateType {LOCKED, UNLOCKED};
```

3.1.2.3.3.3.6 AvailabilityStatusType.

AvailabilityStatusType denotes whether or not the Log is available for use. When true, offDuty indicates the Log is LOCKED (administrative state) or DISABLED (operational state). When true, logFull indicates the Log storage is full.

```
struct AvailabilityStatusType{
     boolean offDuty;
     boolean logFull;
};
```

3.1.2.3.3.3.7 LogFullActionType.

This type specifies the action that the Log should take when its internal buffers become full of data, leaving no room for new records to be written. Wrap indicates that the Log will overwrite the oldest log records with the newest records, as they are written to the Log. Halt indicates that the Log will stop logging when full.

```
enum LogFullActionType (WRAP, HALT);
```

3.1.2.3.3.3.8 RecordIDType.

This type provides the record ID that is assigned to a log record.

```
typedef unsigned long long RecordIDType;
```

3.1.2.3.3.3.9 LogRecordType.

The LogRecordType defines the format of the log records as stored in the Log. The ‘info’ field is the ProducerLogRecord that is written by a client to the Log.

```
struct LogRecordType {
      RecordIDType id;
      LogTimeType time;
```

```
      ProducerLogRecordType info;
      };
```

3.1.2.3.3.3.10 LogRecordSequence.

The LogRecordSequence type defines an unbounded sequence of log records.

```
typedef sequence<LogRecordType> LogRecordSequence;
```

3.1.2.3.3.3.11 ProducerLogRecordSequence Type.

The ProducerLogRecordSequence type defines a sequence of ProducerLogRecordTypes.

```
typedef sequence <ProducerLogRecordType> ProducerLogRecordSequence
```

3.1.2.3.3.4 Attributes.

N/A.

3.1.2.3.3.5 Operations.

3.1.2.3.3.5.1 *getMaxSize*.

3.1.2.3.3.5.1.1 Brief Rationale.

This operation sets the maximum number of bytes that the Log can store.

3.1.2.3.3.5.1.2 Synopsis.

```
unsigned long long getMaxSize();
```

3.1.2.3.3.5.1.3 Behavior.

The *getMaxSize* operation returns the maximum size of the Log measured in number of bytes.

3.1.2.3.3.5.1.4 Returns.

The *getMaxSize* operation shall return the integer number of bytes that the Log is capable of storing.

3.1.2.3.3.5.1.5 Exceptions/Errors.

This operation does not raise any exceptions.

3.1.2.3.3.5.2 *setMaxSize*.

3.1.2.3.3.5.2.1 Brief Rationale.

This operation sets the maximum number of bytes that the Log can store.

3.1.2.3.3.5.2.2 Synopsis.

```
void setMaxSize(unsigned long long size) raises (InvalidParam);
```

3.1.2.3.3.5.2.3 Behavior.

The *setMaxSize* operation shall set the maximum size of the log measured in number of bytes.

3.1.2.3.3.5.2.4 Returns.

This operation does not return a value.

3.1.2.3.3.5.2.5 Exceptions/Errors.

The *setMaxSize* operation shall raise the InvalidParam exception if the size parameter passed in is less than the current size of the Log.

The *setMaxSize* operation shall raise the InvalidParam exception if the input size parameter is greater than the storage space available to the Log.

3.1.2.3.3.5.3 *getCurrentSize*.

3.1.2.3.3.5.3.1 Brief Rationale.

The *getCurrentSize* operation provides the current size of the log storage in bytes.

3.1.2.3.3.5.3.2 Synopsis.

```
unsigned long long getCurrentSize ();
```

3.1.2.3.3.5.3.3 Behavior.

The *getCurrentSize* operation returns the current size of the log storage in bytes.

3.1.2.3.3.5.3.4 Returns.

The *getCurrentSize* operation shall return the current size of the log storage in bytes. (i.e. if the log contains no records, *getCurrentSize* will return a value of 0 (zero).)

3.1.2.3.3.5.3.5 Exceptions/Errors.

This operation does not return any exceptions.

3.1.2.3.3.5.4 *getNumRecords*.

3.1.2.3.3.5.4.1 Brief Rationale.

The *getNumRecords* operation provides the number of records present in the Log.

3.1.2.3.3.5.4.2 Synopsis.

```
unsigned long long getNumRecords ();
```

3.1.2.3.3.5.4.3 Behavior.

The *getNumRecords* operation returns the current number of records contained in the Log.

3.1.2.3.3.5.4.4 Returns.

The *getNumRecords* operation shall return the current number of log records contained in the Log.

3.1.2.3.3.5.4.5 Exceptions/Errors.

This operation does not raise any exceptions.

3.1.2.3.3.5.5 *getLogFullAction*.

3.1.2.3.3.5.5.1 Brief Rationale.

The *getLogFullAction* operation provides the action taken when the Log becomes full.

3.1.2.3.3.5.5.2 Synopsis.

```
LogFullActionType getLogFullAction();
```

3.1.2.3.3.5.5.3 Behavior.

The *getLogFullAction* operation returns the action that will be taken when the maximum size of the Log has been reached.

3.1.2.3.3.5.5.4 Returns.

The *getLogFullAction* operation shall return the Log's log full action setting.

3.1.2.3.3.5.5.5 Exceptions/Errors.

This operation does not return any exceptions.

3.1.2.3.3.5.6 *setLogFullAction.*

3.1.2.3.3.5.6.1 Brief Rationale.

The *setLogFullAction* operation is used to configure the Log to either WRAP or HALT when the Log becomes full.

3.1.2.3.3.5.6.2 Synopsis.

```
void setLogFullAction(in LogFullActionType action)
```

3.1.2.3.3.5.6.3 Behavior.

The *setLogFullAction* operation shall set the action taken when the maximum size of the Log has been reached.

3.1.2.3.3.5.6.4 Returns.

This operation does not return a value.

3.1.2.3.3.5.6.5 Exceptions/Errors.

This operation does not return any exceptions.

3.1.2.3.3.5.7 *getAvailabilityStatus.*

3.1.2.3.3.5.7.1 Brief Rationale.

The *getAvailabilityStatus* operation is used to read the availability status of the Log.

3.1.2.3.3.5.7.2 Synopsis.

```
AvailabilityStatusType getAvailabilityStatus ();
```

3.1.2.3.3.5.7.3 Behavior.

The *getAvailabilityStatus* operation returns a structure that reflects the availability status of the Log. See the description of the AvailabilityStatusType in 3.1.2.3.3.3.6.

3.1.2.3.3.5.7.4 Returns.

The *getAvailabilityStatus* operation shall return the current availability status of the Log.

3.1.2.3.3.5.7.5 Exceptions/Errors.

This operation does not raise any exceptions.

3.1.2.3.3.5.8 *getAdministrativeState.*

3.1.2.3.3.5.8.1 Brief Rationale.

The *getAdministrativeState* is used to read the administrative state of the Log.

3.1.2.3.3.5.8.2 Synopsis.

```
AdministrativeStateType getAdministrativeState();
```

3.1.2.3.3.5.8.3 Behavior.

The *getAdministrativeState* operation returns the administrative state of the Log. See the description of the AdministrativeStateType in 3.1.2.3.3.3.5.

3.1.2.3.3.5.8.4 Returns.

The *getAdministrativeState* operation shall return the current administrative state of the Log.

3.1.2.3.3.5.8.5 Exceptions/Errors.

This operation does not raise any exceptions.

3.1.2.3.3.5.9 *setAdministrativeState*.

3.1.2.3.3.5.9.1 Brief Rationale.

The *setAdministrativeState* operation provides write access to the administrative state value.

3.1.2.3.3.5.9.2 Synopsis.

```
void setAdministrativeState(in AdministrativeStateType state);
```

3.1.2.3.3.5.9.3 Behavior.

The *setAdministrativeState* operation shall set the administrative state of the Log.

3.1.2.3.3.5.9.4 *Returns*.

This operation does not return a value.

3.1.2.3.3.5.9.5 Exceptions/Errors.

This operation does not raise any *exceptions*.

3.1.2.3.3.5.10 *getOperationalState*.

3.1.2.3.3.5.10.1 Brief Rationale.

The *getOperationalState* operation returns the operational state of the Log.

3.1.2.3.3.5.10.2 Synopsis.

```
OperationalStateType getOperationalState();
```

3.1.2.3.3.5.10.3 Behavior.

The *getOperationalState* operation returns the operational state of the Log. See the description of OperationalStateType in 3.1.2.3.3.3.4.

3.1.2.3.3.5.10.4 Returns.

The *getOperationalState* operation shall return the current operational state of the Log.

3.1.2.3.3.5.10.5 Exceptions/Errors.

This operation does not raise any exceptions.

3.1.2.3.3.5.11 *writeRecords.*

3.1.2.3.3.5.11.1 Brief Rationale.

The *writeRecords* operation provides the method for writing log records to the Log. The operation is defined as oneway to minimize client overhead while writing to the Log.

3.1.2.3.3.5.11.2 Synopsis.

```
oneway void writeRecords(in ProducerLogRecordSequence records);
```

3.1.2.3.3.5.11.3 Behavior.

The *writeRecords* operation shall add each log record supplied in the records parameter to the *Log*. When there is insufficient storage to add one of the supplied log records to the Log, and theLogFullAction is set to HALT, the *writeRecords* method shall set the availability status logFull state to true (e.g. if 3 records are provided in the records parameter, and while trying to write the second record to the log, the record will not fit, then the log is considered to be full therefore the second and third records will not be stored in the log but the first record would have been successfully stored.).

The *writeRecords* operation shall write the current time to the *time* field of the LogRecord in the format defined by the standard POSIX type *struct tm*.

The *writeRecords* operation shall assign a unique record ID to the id field of the LogRecord.

Log records accepted for storage by the *writeRecords* shall be available for retrieval in the order received.

3.1.2.3.3.5.11.4 Returns.

This operation does not return a value.

3.1.2.3.3.5.11.5 Exceptions/Errors.

This operation does not raise any exceptions.

3.1.2.3.3.5.12 *getRecordIdFromTime*.

3.1.2.3.3.5.12.1 Brief Rationale.

The *getRecordIdFromTime* operation is used to get the record ID of the first record in the Log with a time-stamp that is greater than or equal to the time specified in the parameter.

3.1.2.3.3.5.12.2 Synopsis.

```
RecordIDType getRecordIdFromTime (in LogTimeType fromTime);
     raises (LogEmpty)
```

3.1.2.3.3.5.12.3 Behavior.

The *getRecordIdFromTime* operation returns the record ID of the first record in the log with a time-stamp that is greater than, or equal to, the time specified in the fromTime parameter. If the Log is empty, the LogEmpty exception is raised, and the returned record ID is invalid.

3.1.2.3.3.5.12.4 Returns.

The *getRecordIdFromTime* operation shall return the record ID of the first record in the log with a time-stamp that is greater than, or equal to, the time specified in the fromTime parameter. The *getRecordIdFromTime* operation shall return zero if no record exists in the log with a time-stampthat is greater than or equal to the time specified in the fromTime parameter.

3.1.2.3.3.5.12.5 Exceptions/Errors.

When the Log is empty, the *getRecordIdFromTime* operation shall raise the LogEmpty exception.

3.1.2.3.3.5.13 *retrieveById*.

3.1.2.3.3.5.13.1 Brief Rationale.

The *retrieveById* operation is used to get a specified number of records from a Log.

3.1.2.3.3.5.13.2 Synopsis.

```
LogRecordSequence retrieveById (inout RecordIDType currentId, in unsigned
long howMany);
```

3.1.2.3.3.5.13.3 Behavior.

The *retrieveById* operation returns a list of log records that begins with the record specified bythe currentID parameter and contains less than or equal to the number of records specified in the howMany parameter.

The *retrieveById* operation shall set the inout parameter currentId to the LogRecord Id of next record after the last record in the log record sequence returned.

3.1.2.3.3.5.13.4 Returns.

The *retrieveById* operation shall return a log record sequence that begins with the record specified by the currentId parameter. The number of records in the log record sequence returned by the *retrieveById* operation shall be equal to the number of records specified by the howMany parameter, or the number of records available if the number of records specified by the howMany parameter cannot be met. If the record specified by currentId does not exist, the *retrieveById* operation shall return an empty list of log records. If the Log is empty, the *retrieveById* operation shall return an empty list of log records.

3.1.2.3.3.5.13.5 Exceptions/Errors.

This operation does not raise any exceptions.

3.1.2.3.3.5.14 *clearLog*.

3.1.2.3.3.5.14.1 Brief Rationale.

The *clearLog* operation provides the method for removing all of the log records from the Log.

3.1.2.3.3.5.14.2 Synopsis.

```
void clearLog ();
```

3.1.2.3.3.5.14.3 Behavior.

The *clearLog* operation shall delete all records from the Log. The *clearLog* operation shall set the current size of the Log storage to zero. The *clearLog* operation shall set the current number of records in the Log to zero. The *clearLog* operation shall set the logFull availability status element to false.

3.1.2.3.3.5.14.4 Returns.

This operation does not return a value.

3.1.2.3.3.5.14.5 Exceptions/Errors.

This operation does not raise any exceptions.

3.1.2.3.3.5.15 *destroy*.

3.1.2.3.3.5.15.1 Brief Rationale.

The *destroy* operation provides a means by which an instantiated Log may be torn down.

3.1.2.3.3.5.15.2 Synopsis.

```
void destroy ();
```

3.1.2.3.3.5.15.3 Behavior.

The *destroy* operation shall release all internal memory and/or storage allocated by the Log. The *destroy* operation shall tear down the component (i.e. released from the CORBA environment).

3.1.2.3.3.5.15.4 Returns.

This operation does not return a value.

3.1.2.3.3.5.15.5 Exceptions/Errors.

This operation does not raise any exception.

3.1.3 Core Framework.

The CF specification includes a detailed description of the purpose of each interface, the purpose of each supported operation within the interface, and interface class diagrams to support these descriptions. The corresponding IDL for the CF can be found in Appendix C.

Figure 3-3 depicts the key elements of the CF and the IDL relationships between these elements. A *DomainManager* component manages the software *Applications*, *ApplicationFactories*, hardware devices (represented by software *Devices*) and *DeviceManagers* within the system. An *Application* is a type of *Resource* and consists of one to many software *Resources*. Some of the software *Resources* may directly control the system's internal hardware devices; these *Resources* are logical *Devices* which implement the *Device*, *LoadableDevice,* or *ExecutableDevice* interfaces. (For example, a *ModemDevice* may provide direct control of a modem hardware device such as a Field Programmable Gate Array (FPGA) or an Application Specific Integrated Circuit (ASIC). An *I/ODevice* may operate as a device driver to provide external access to the system.) Other software *Resources* have no direct relationship with a hardware device, but perform application services for the user. (For example, a *NetworkResource* may perform a network layer function. A *WaveformLinkResource* may perform a waveform specific link layer service.) Each *Resource* can potentially communicate with other *Resources*. *Devices* are allocated to one or more hardware devices by the *DomainManager* based upon various factors including the hardware devices that the *DeviceManager* knows about, the current availability of hardware devices, the behavior rules of a *Resource*, and the loading requirements of the *Resource*.

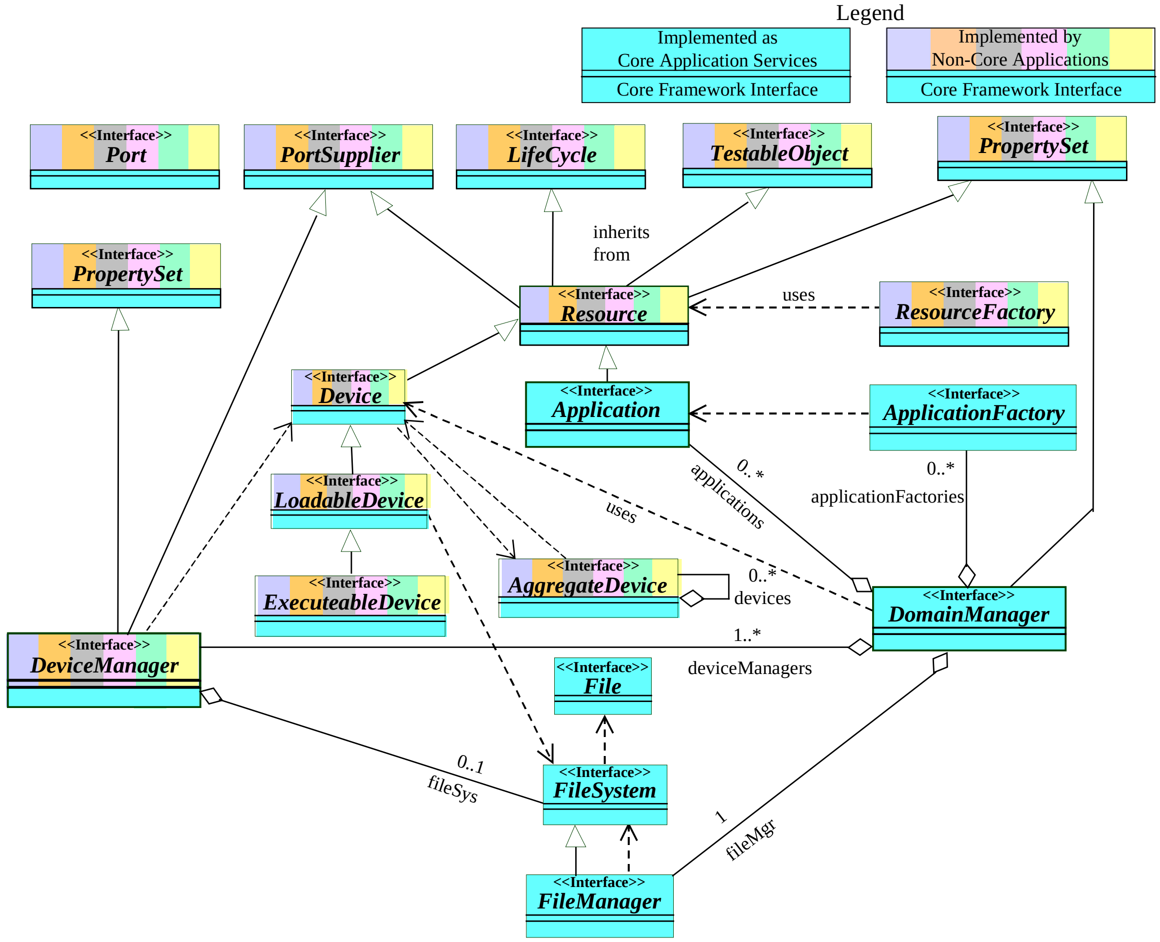

Figure 3-3. Core Framework IDL Relationships

The *Resources* being managed by the *DomainManager* are CORBA objects implementing the *Resource* interface. Some *Resources* may be dependent on other *Resources*. This interface provides a consistent way of creating up and tearing down any *Resource* within the system. These resources can be created by using a *ResourceFactory* interface or by the *Device* interfaces (*Device, LoadableDevice, or ExecutableDevice*).

The file service interfaces (*FileManager*, *FileSystem*, and *File*) are used for installation and removal of application files within the system, and for loading and unloading application files on the various processors that the *Devices* execute upon.

3.1.3.1 Base Application Interfaces.

Base Application Interfaces are defined by the Core Framework requirements and implemented by application developers; see 3.2 for Application requirements.

3.1.3.1.1 *Port*.

3.1.3.1.1.1 Description.

This interface provides operations for managing associations between ports. The *Port* interface UML is depicted in Figure 3-4. An application defines a specific *Port* type by specifying an interface that inherits the *Port* interface. An application establishes the operations for transferring data and control. The application also establishes the meaning of the data and control values. Examples of how applications may use ports in different ways include: push or pull, synchronous or asynchronous, mono- or bi-directional, or whether to use flow control (e.g., pause, start, stop).

The nature of *Port* fan-in, fan-out, or one-to-one is component dependent.

Note 1: The CORBA specification defines only a minimum size for each basic IDL type. The actual size of the data type is dependent on the language (defined in the language mappings) as well as the Central Processing Unit (CPU) architecture used. By using these CORBA basic data types, portability is maintained between components implemented in differing CPU architectures and languages.

Note 2: How components' ports are connected is described in the software assembly descriptor (SAD) file of the Domain Profile (3.1.3.4).

3.1.3.1.1.2 UML.

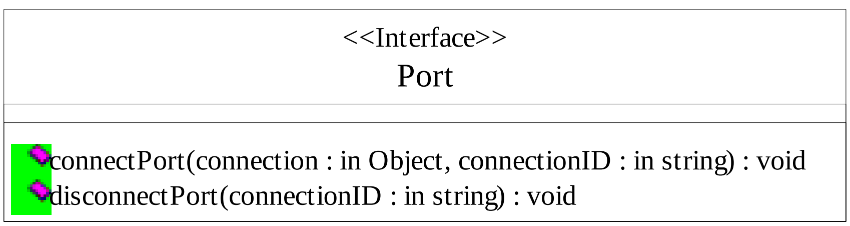

Figure 3-4. *Port* Interface UML

3.1.3.1.1.3 Types.

3.1.3.1.1.3.1 InvalidPort.

The InvalidPort exception indicates one of the following errors has occurred in the specification of a *Port* association:

1. errorCode 1 means the *Port* component is invalid (unable to narrow object reference) or illegal object reference,
2. errorCode 2 means the *Port* name is not found (not used by this *Port*).

```
exception InvalidPort { unsigned short errorCode, string msg };
```

3.1.3.1.1.3.2 OccupiedPort.

The OccupiedPort exception indicates the *Port* is unable to accept any additional connections.

```
exception OccupiedPort {};
```

3.1.3.1.1.4 Attributes.

N/A.

3.1.3.1.1.5 Operations.

3.1.3.1.1.5.1 *connectPort*.

3.1.3.1.1.5.1.1 Brief Rationale.

Applications require the *connectPort* operation to establish associations between *Ports*. *Ports* provide channels through which data and/or control pass.

The *connectPort* operation provides half of a two-way association; therefore two calls are required to create a two-way association.

3.1.3.1.1.5.1.2 Synopsis.

```
void connectPort(in Object connection, in string connectionID) raises
(InvalidPort, OccupiedPort);
```

3.1.3.1.1.5.1.3 Behavior.

The connect*Port* operation shall make a connection to the component identified by the input parameters.

A port may support several connections. The input connectionID is a unique identifier to be used by *disconnectPort* when breaking this specific connection.

3.1.3.1.1.5.1.4 Returns.

This operation does not return a value.

3.1.3.1.1.5.1.5 Exceptions/Errors.

The *connectPort* operation shall raise the InvalidPort exception when the input connection parameter is an invalid connection for this Port.

The *connectPort* operation shall raise the OccupiedPort exception when unable to accept the connections because the *Port* is already fully occupied.

3.1.3.1.1.5.2 *disconnectPort*.

3.1.3.1.1.5.2.1 Brief Rationale.

Applications require the *disconnectPort* operation in order to allow consumer/producer data components to disassociate themselves from their counterparts (consumer/producer).

3.1.3.1.1.5.2.2 Synopsis.

```
void disconnectPort (in string connectionID) raises (InvalidPort);
```

3.1.3.1.1.5.2.3 Behavior.

The *disconnectPort* operation shall break the connection to the component identified by the input parameters.

3.1.3.1.1.5.2.4 Returns.

This operation does not return a value.

3.1.3.1.1.5.2.5 Exceptions/Errors.

The *disconnectPort* operation shall raise the InvalidPort exception when the name passed to *disconnectPort* is not connected with the *Port* component.

3.1.3.1.2 *LifeCycle*.

3.1.3.1.2.1 Description.

The *LifeCycle* interface defines the generic operations for initializing or releasing instantiated component-specific data and/or processing elements. The *LifeCycle* interface UML is depicted in Figure 3-5.

3.1.3.1.2.2 UML.

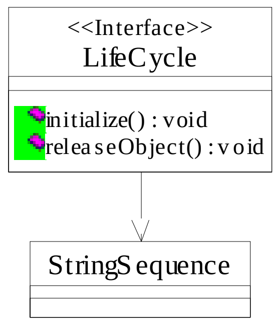

Figure 3-5. *LifeCycle* Interface UML

3.1.3.1.2.3 Types.

3.1.3.1.2.3.1 InitializeError.

The InitializeError exception indicates an error occurred during component initialization. The message is component-dependent, providing additional information describing the reason why the error occurred.

```
exception InitializeError { StringSequence errorMessage; };
```

3.1.3.1.2.3.2 ReleaseError.

The ReleaseError exception indicates an error occurred during component *releaseObject*. The message is component-dependent, providing additional information describing the reason why the error occurred.

```
exception ReleaseError { StringSequence errorMessage; };
```

3.1.3.1.2.4 Attributes.

N/A.

3.1.3.1.2.5 Operations.

3.1.3.1.2.5.1 *initialize*.

3.1.3.1.2.5.1.1 Brief Rationale.

The purpose of the *initialize* operation is to provide a mechanism to set a component to a known initial state. (For example, data structures may be set to initial values, memory may be allocated, hardware devices may be configured to some state, etc.)

3.1.3.1.2.5.1.2 Synopsis.

```
void initialize() raises (InitializeError);
```

3.1.3.1.2.5.1.3 Behavior.

Initialization behavior is implementation dependent.

3.1.3.1.2.5.1.4 Returns.

This operation does not return a value.

3.1.3.1.2.5.1.5 Exceptions/Errors.

The *initialize* operation shall raise an InitializeError exception when an initialization error occurs.

3.1.3.1.2.5.2 *releaseObject*.

3.1.3.1.2.5.2.1 Brief Rationale.

The purpose of the *releaseObject* operation is to provide a means by which an instantiated component may be torn down.

3.1.3.1.2.5.2.2 Synopsis.

```
void releaseObject() raises (ReleaseError);
```

3.1.3.1.2.5.2.3 Behavior.

The *releaseObject* operation shall release all internal memory allocated by the component during the life of the component. The *releaseObject* operation shall tear down the component (i.e. released from the CORBA environment). The *releaseObject* operation shall release components from the OE.

3.1.3.1.2.5.2.4 Returns.

This operation does not return a value.

3.1.3.1.2.5.2.5 Exceptions/Errors.

The *releaseObject* operation shall raise a ReleaseError exception when a release error occurs.

3.1.3.1.3 *TestableObject*.

3.1.3.1.3.1 Description.

The *TestableObject* interface defines a set of operations that can be used to test component implementations. The *TestableObject* interface UML is depicted in Figure 3-6.

3.1.3.1.3.2 UML.

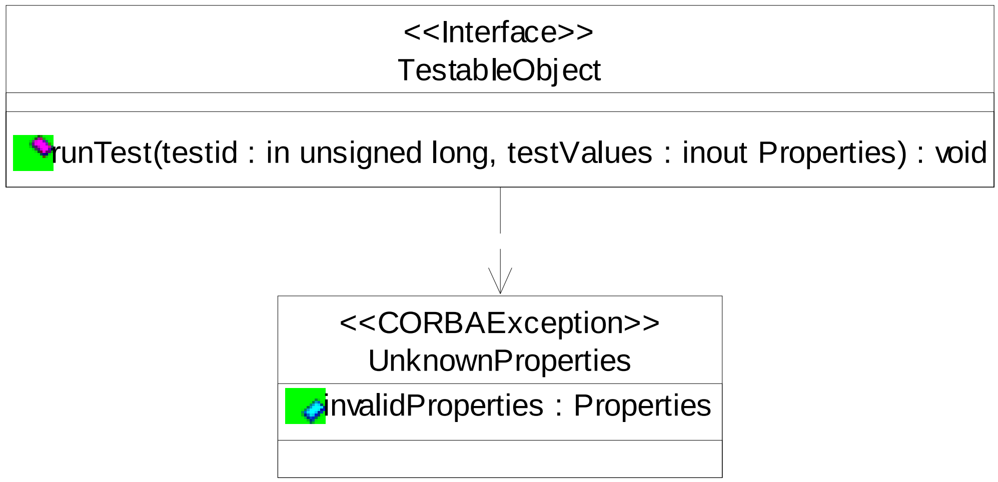

Figure 3-6. *TestableObject* Interface UML

3.1.3.1.3.3 Types.

3.1.3.1.3.3.1 UnknownTest.

The UnknownTest exception indicates the requested testid for a test to be performed is not known by the component.

```
exception UnknownTest {};
```

3.1.3.1.3.4 Attributes.

N/A.

3.1.3.1.3.5 Operations.

3.1.3.1.3.5.1 *runTest*.

3.1.3.1.3.5.1.1 Brief Rationale.

The *runTest* operation allows components to be “blackbox” tested. This allows Built-In Test (BIT) to be implemented and this provides a means to isolate faults (both software and hardware) within the system.

3.1.3.1.3.5.1.2 Synopsis.

```
void runTest(in unsigned long testid, inout Properties testValues)raises
(UnknownTest, UnknownProperties);
```

3.1.3.1.3.5.1.3 Behavior.

The *runTest* operation shall use the testid parameter to determine which of its predefined test implementations should be performed. The testValues parameter CF Properties (id/value pair(s)) shall be used to provide additional information to the implementation-specific test to be run. The *runTest* operation shall return the result(s) of the test in the testValues parameter.

Tests to be implemented by a component are component-dependent and are specified in the component's Properties Descriptor. Valid testid(s) and both input and ouput testValues (properties) for the *runTest* operation shall at a minimum be test properties defined in the *properties test* element of the component's Properties Descriptor (refer to Appendix D Domain Profile). The testid parameter corresponds to the XML attribute testid of the property element test in a propertyfile.

A CF UnknownProperties exception is raised by the runTest operation all inputValues properties shall be validated (i.e., test properties defined in the propertyfile(s) referenced in the component's SPD).

The *runTest* operation shall not execute any testing when the input testid or any of the input testValues are not known by the component or are out of range.

3.1.3.1.3.5.1.4 Returns.

This operation does not return a value.

3.1.3.1.3.5.1.5 Exceptions/Errors.

The *runTest* operation shall raise the UnknownTest exception when there is no underlying test implementation that is associated with the input testid given.

The *runTest* operation shall raise the CF UnknownProperties exception when the input parameter testValues contains any CF DataTypes that are not known by the component's test implementation or any values that are out of range for the requested test. The exception parameter invalidProperties shall contain the invalid inputValues properties id(s) that are not known by the component or the value(s) are out of range.

3.1.3.1.4 *PortSupplier*.

3.1.3.1.4.1 Description.

This interface provides the *getPort* operation for those components that provide ports.

3.1.3.1.4.2 UML.

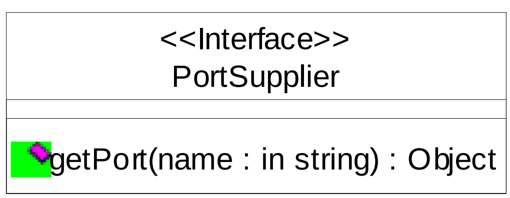

Figure 3-7. *PortSupplier* Interface UML

3.1.3.1.4.3 Types.

3.1.3.1.4.3.1 UnknownPort.

The UnknownPort exception is raised if an undefined port is requested.

```
exception UnknownPort { };
```

3.1.3.1.4.4 Attributes.

N/A.

3.1.3.1.4.5 Operations.

3.1.3.1.4.5.1 *getPort*.

3.1.3.1.4.5.1.1 Brief Rationale.

The *getPort* operation provides a mechanism to obtain a specific consumer or producer *Port*. A *PortSupplier* may contain zero-to-many consumer and producer port components. The exact number is specified in the component's Software Profile SCD (section 3.1.3.4). These *Ports* can be either push or pull types. Multiple input and/or output ports provide flexibility for *PortSuppliers* that must manage varying priority levels and categories of incoming and outgoing messages, provide multi-threaded message handling, or other special message processing.

3.1.3.1.4.5.1.2 Synopsis.

```
Object getPort(in string name) raises ( UnknownPort );
```

3.1.3.1.4.5.1.3 Behavior.

The *getPort* operation returns the object reference to the named port as stated in the *component's* SCD.

3.1.3.1.4.5.1.4 Returns.

The *getPort* operation shall return the CORBA object reference that is associated with the input port name.

3.1.3.1.4.5.1.5 *Exceptions*/Errors.

The *getPort* operation shall raise an UnknownPort exception if the port name is invalid.

3.1.3.1.5 *PropertySet*.

3.1.3.1.5.1 Description.

The *PropertySet* interface defines *configure* and *query* operations to access component properties/attributes. The *PropertySet* interface UML is depicted in Figure 3-8.

3.1.3.1.5.2 UML.

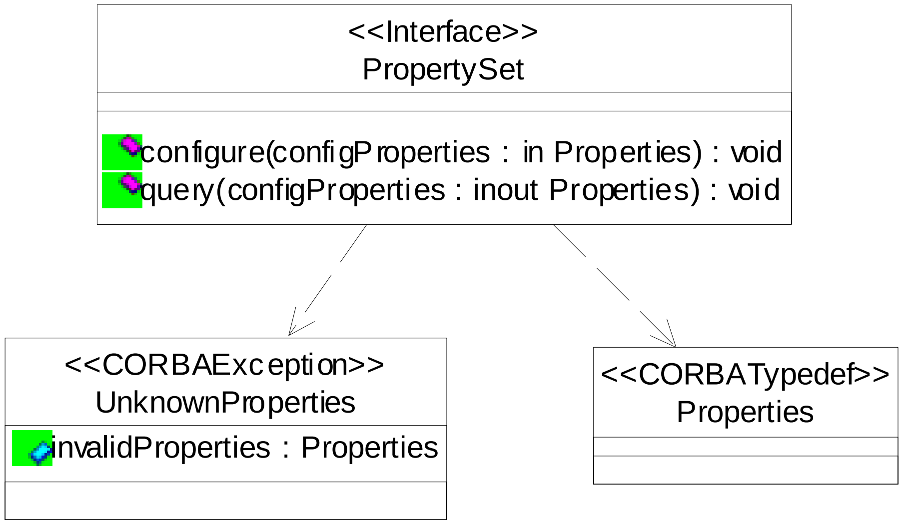

Figure 3-8. *PropertySet* Interface UML

3.1.3.1.5.3 Types.

N/A.

3.1.3.1.5.3.1 InvalidConfiguration.

The InvalidConfiguration indicates the configuration of a component has failed (no configuration at all was done). The message is component-dependent, providing additional information describing the reason why the error occurred.. The invalidProperties returned indicate the properties that were invalid.

```
exception InvalidConfiguration { string msg; Properties invalidProperties};
```

3.1.3.1.5.3.2 PartialConfiguration.

The PartialConfiguration exception indicates the configuration of a Component was partiallysuccessful. The invalidProperties returned indicate the properties that were invalid.

```
exception PartialConfiguration { Properties invalidProperties};
```

3.1.3.1.5.4 Attributes.

N/A.

3.1.3.1.5.5 Operations.

3.1.3.1.5.5.1 *configure*.

3.1.3.1.5.5.1.1 Brief Rationale.

The *configure* operation allows id/value pair configuration properties to be assigned to components implementing this interface.

3.1.3.1.5.5.1.2 Synopsis.

```
void configure(in Properties configProperties) raises (InvalidConfiguration,
PartialConfiguration);
```

3.1.3.1.5.5.1.3 Behavior.

The *configure* operation shall assign values to the properties as indicated in the configProperties argument. Valid properties for the *configure* operation shall at a minimum be the configure readwrite and writeonly properties referenced in the component's SPD.

3.1.3.1.5.5.1.4 Returns.

This operation does not return a value.

3.1.3.1.5.5.1.5 Exceptions/Errors.

The *configure* operation shall raise a PartialConfiguration exception when some configuration properties were successfully set and some configuration properties were not successfully set.

The *configure* operation shall raise an InvalidConfiguration exception when a configuration error occurs that prevents any property configuration on the component.

3.1.3.1.5.5.2 *query*.

3.1.3.1.5.5.2.1 Brief Rationale.

The *query* operation allows a component to be queried to retrieve its properties.

3.1.3.1.5.5.2.2 Synopsis.

```
void query(inout Properties configProperties) raises (UnknownProperties);
```

3.1.3.1.5.5.2.3 Behavior.

If the configProperties are zero size then, the *query* operation shall return all component properties. If the configProperties are not zero size, then the *query* operation shall return only those id/value pairs specified in the configProperties. Valid properties for the *query* operation shall at a minimum be the configure readwrite and readonly properties, and allocation properties that have an action value of "external" as referenced in the component's SPD.

3.1.3.1.5.5.2.4 Returns.

This operation does not return a value.

3.1.3.1.5.5.2.5 Exceptions/Errors.

The *query* operation shall raise the CF UnknownProperties exception when one or more properties being requested are not known by the component.

3.1.3.1.6 *Resource*.

3.1.3.1.6.1 Description.

The *Resource* interface provides a common API for the control and configuration of a software component. The *Resource* interface UML is depicted in Figure 3-9.

The Resource interface inherits from the *LifeCycle*, *PropertySet*, *TestableObject*, and *PortSupplier* interfaces.

The inherited *LifeCycle*, *PropertySet*, *TestableObject*, and *PortSupplier* interface operations are documented in their respective sections of this document.

The *Resource* interface may also be inherited by other application interfaces as described in the Software Profile's Software Component Descriptor (SCD) file (see 3.1.3.4).

3.1.3.1.6.2 UML.

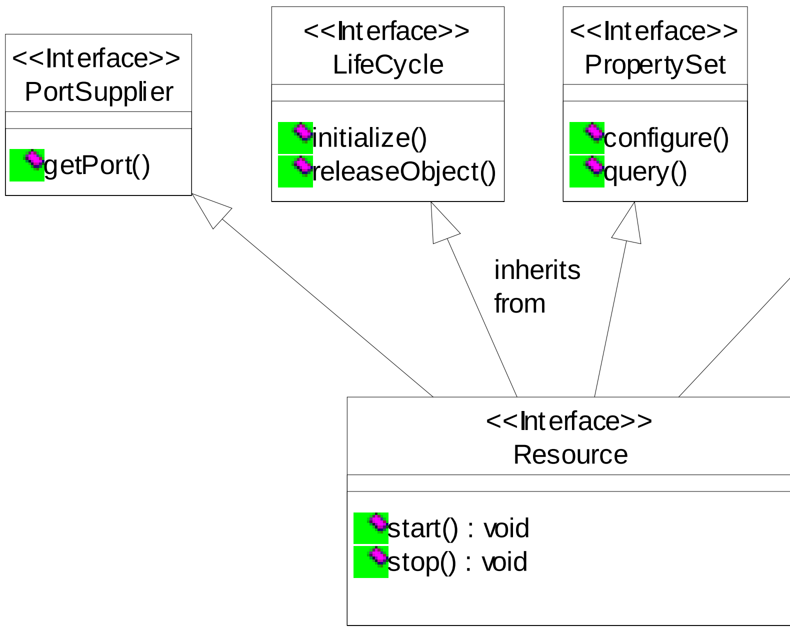

Figure 3-9. *Resource* Interface UML

3.1.3.1.6.3 Types.

3.1.3.1.6.3.1 UnknownPort.

The UnknownPort exception is raised if an undefined port is requested.

```
exception UnknownPort { };
```

3.1.3.1.6.3.2 StartError.

The StartError exception indicates that an error occurred during an attempt to start the *Resource*. The message is component-dependent, providing additional information describing the reason for the error and the severity of the error.

```
exception StartError { string msg };
```

3.1.3.1.6.3.3 StopError.

The StopError exception indicates that an error occurred during an attempt to stop the *Resource*. The message is component-dependent, providing additional information describing the reason for the error and the severity of the error.

```
exception StopError { string msg };
```

3.1.3.1.6.4 Attributes.

N/A.

3.1.3.1.6.5 Operations.

3.1.3.1.6.5.1 *stop*.

3.1.3.1.6.5.1.1 Brief Rationale.

The *stop* operation is provided to command a *Resource* implementing this interface to stop internal processing.

3.1.3.1.6.5.1.2 Synopsis.

```
void stop()raises (StopError);
```

3.1.3.1.6.5.1.3 Behavior.

The *stop* operation shall disable all current operations and put the *Resource* in a non-operating condition. Subsequent *configure*, *query*, and *start* operations are not inhibited by the *stop* operation.

3.1.3.1.6.5.1.4 Returns.

This operation does not return a value.

3.1.3.1.6.5.1.5 Exceptions/Errors.

The *stop* operation shall raise the StopError exception if an error occurs while stopping the resource.

3.1.3.1.6.5.2 *start*.

3.1.3.1.6.5.2.1 Brief Rationale.

The *start* operation is provided to command a *Resource* implementing this interface to start internal processing.

3.1.3.1.6.5.2.2 Synopsis.

```
void start()raises (StartError);
```

3.1.3.1.6.5.2.3 Behavior.

The *start* operation puts the *Resource* in an operating condition.

3.1.3.1.6.5.2.4 Returns.

This operation does not return a value.

3.1.3.1.6.5.2.5 Exceptions/Errors.

The *start* operation shall raise the StartError exception if an error occurs while starting the resource.

3.1.3.1.6.5.3 *getPort*.

3.1.3.1.6.5.3.1 Brief Rationale.

The *getPort* operation provides a mechanism to obtain a specific consumer or producer *Port*. A *Resource* may contain zero-to-many consumer and producer port components. The exact number is specified in the component's Software Profile SCD (section 3.1.3.4). These *Ports* can be either push or pull types. Multiple input and/or output ports provide flexibility for *Applications* and *Resources* that must manage varying priority levels and categories of incoming and outgoing messages, provide multi-threaded message handling, or other special message processing.

3.1.3.1.6.5.3.2 Synopsis.

```
Object getPort(in string name) raises ( UnknownPort );
```

3.1.3.1.6.5.3.3 Behavior.

The *getPort* operation shall return the object reference to the named port as stated in the *Resource's* SCD.

3.1.3.1.6.5.3.4 Returns.

The *getPort* operation returns the CORBA object reference that is associated with the input port name.

3.1.3.1.6.5.3.5 Exceptions/Errors.

The *getPort* operation shall raise an UnknownPort exception if the port name is invalid.*ResourceFactory*.

3.1.3.1.7.1 Description.

A *ResourceFactory* is used to create and tear down a *Resource*. The *ResourceFactory* interface is designed after the Factory Design Patterns. The *ResourceFactory* interface UML is depicted in Figure 3-10. The factory mechanism provides client-server isolation among *Resources* (e.g., Network, Link, Modem, I/O, etc.) and provides an industry standard mechanism of obtaining a *Resource* without knowing its identity. An application is not required to use *ResourceFactories* to obtain, create, or tear down resources. A Software Profile will determine which application *ResourceFactories* are to be used by the *ApplicationFactory*.

3.1.3.1.7.2 UML.

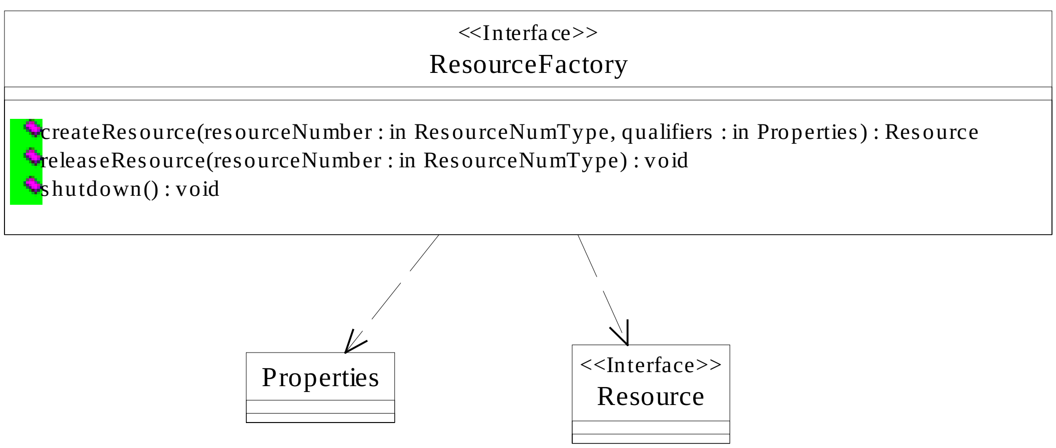

Figure 3-10. *ResourceFactory* Interface UML

3.1.3.1.7.3 Types.

3.1.3.1.7.3.1 ResourceNumType.

This type defines the identity of a *Resource* created by the *ResourceFactory*.

```
typedef unsigned short ResourceNumType.
```

3.1.3.1.7.3.2 InvalidResourceNumber.

The InvalidResourceNumber exception indicates the resourceNumber does not exist in the *Factory*.

```
exception InvalidResourceNumber { };
```

3.1.3.1.7.3.3 ShutdownFailure.

The ShutdownFailure exception indicates that the shutdown method failed to release the *ResourceFactory* from the CORBA environment due to the fact the *Factory* still contains *Resources*. The message is component-dependent, providing additional information describing why the shutdown failed.

```
exception ShutdownFailure { string msg };
```

3.1.3.1.7.4 Attributes.

N/A.

3.1.3.1.7.5 Operations.

3.1.3.1.7.5.1 *createResource*.

3.1.3.1.7.5.1.1 Brief Rationale.

The *createResource* operation provides the capability to create *Resources* in the same process space as the *ResourceFactory* or to return a *Resource* that has already been created. This behavior is an alternative approach for creating a *Resource* to the *Device execute* operations.

3.1.3.1.7.5.1.2 Synopsis.

The resourceNumber is the identifier for *Resource*. The qualifiers are parameter values used by the *ResourceFactory* in creation of the Resource. The *ApplicationFactory* can determine the values to be supplied for the qualifiers from the description in the *ResourceFactory's* Software Profile. The qualifiers may be used to identify, for example, specific subtypes of *Resources* created by a *ResourceFactory*.

```
Resource createResource (in ResourceNumType resourceNumber, in Properties
qualifiers);
```

3.1.3.1.7.5.1.3 Behavior.

If no *Resource* exists for the given resourceNumber, the *createResource* operation shall create a *Resource*. If the *Resource* already exists, that *Resource* is returned. The*createResource* operation shall assign the given resourceNumber to a new *Resource* and set a reference count to one or, in the case that the *Resource* already exists, increment the count by *one*. The reference count is used to indicate the number of times that a specific *Resource* reference has been given to requesting clients. This ensures that the *ResourceFactory* does not release a *Resource* that has a reference count greater than 0. (Multiple clients could request the release of the *Resource* after obtaining a reference to the *Resource*).

3.1.3.1.7.5.1.4 Returns.

The *createResource* operation shall return a reference to the created *Resource* or the existing *Resource*. The *createResource* operation shall return a nil CORBA component reference when the operation is unable to create or find the *Resource*.

3.1.3.1.7.5.1.5 Exceptions/Errors.

This operation does not raise any exceptions.

3.1.3.1.7.5.2 *releaseResource*.

3.1.3.1.7.5.2.1 Brief Rationale.

In CORBA there is client side and server side representation of a *Resource*. The *releaseResource* operation provides the mechanism of releasing the *Resource* in the CORBA environment on the server side when all clients are through with a specific *Resource*. The client still has to release its client side reference of the *Resource*.

3.1.3.1.7.5.2.2 Synopsis.

```
void releaseResource(in ResourceNumType resourceNumber) raises
{InvalidResourceNumber);
```

3.1.3.1.7.5.2.3 Behavior.

The *releaseResource* operation shall decrement the reference count for the specified resource, as indicated by the resourceNumber. The *releaseResource* operation shall make the Resource nolonger avaliable (i.e. it is released from the CORBA environment) when the *Resource's* referencecount is zero.

3.1.3.1.7.5.2.4 Returns.

This operation does not return a value.

3.1.3.1.7.5.2.5 Exceptions/Errors.

The *releaseResource* operation shall raise the InvalidResourceNumber exception if an invalid resourceNumber is received.

3.1.3.1.7.5.3 *shutdown*.

3.1.3.1.7.5.3.1 Brief Rationale.

In CORBA there is client side and server side representation of a *ResourceFactory*. The *shutdown* operation provides the mechanism for releasing the *ResourceFactory* from the CORBA environment on the server side. The client has the responsibility to release its client side reference of the *ResourceFactory*.

3.1.3.1.7.5.3.2 Synopsis.

```
void shutdown()raises {ShutdownFailure);
```

3.1.3.1.7.5.3.3 Behavior.

The *shutdown* operation shall result in the *ResourceFactory* being unavailable to any subsequent calls to its object reference (i.e. it is released from the CORBA environment).

3.1.3.1.7.5.3.4 Returns.

This operation does not return a value.

3.1.3.1.7.5.3.5 Exceptions/Errors.

This operation does not raise any exceptions.

3.1.3.2 Framework Control Interfaces.

"Framework control within a Domain is accomplished by domain management, *device*, anddevice management interfaces.

The Domain Management interfaces are *Application*, *ApplicationFactory,* and *DomainManager*. These interfaces manage the registration and unregistration of applications, devices, and devicemanagers within the domain and the controlling of applications within the domain. Theimplementation of the *Application*, *ApplicationFactory*, and *DomainManager* interfaces arecoupled together and must be delivered together as a complete domain managementimplementation and service.

The device interfaces are for the implementation and management of logical *Devices* within thedomain. The devices within the domain can be simple devices with no loadable, executable, oraggregate device behavior, or devices with a combination of these behaviors. The device interfaces are *Device, LoadableDevice, ExecutableDevice, and AggregateDevice*.

Device management is accomplished by the *DeviceManager* interface. The *DeviceManager* isresponsible for creation of logical *Devices* and launching service applications on these logical *Devices*.

Framework Control Interfaces shall be implemented using the CF IDL presented in AppendixC.

3.1.3.2.1 *Application.*

3.1.3.2.1.1 Description.

The *Application* class provides the interface for the control, configuration, and status of an instantiated application in the domain.

The *Application* interface class inherits the IDL interface of *Resource*. A created application instance may contain *Resource* components and/or non-CORBA components. The *Application* interface UML is depicted in Figure 3-11.

The *Application* interface *releaseObject* operation provides the interface to release the computing resources allocated during the instantiation of the Application, and de-allocate the devices associated with Application instance.

An instance of an *Application* is returned by the *create* operation of an instance of the *ApplicationFactory* class.

3.1.3.2.1.2 UML.

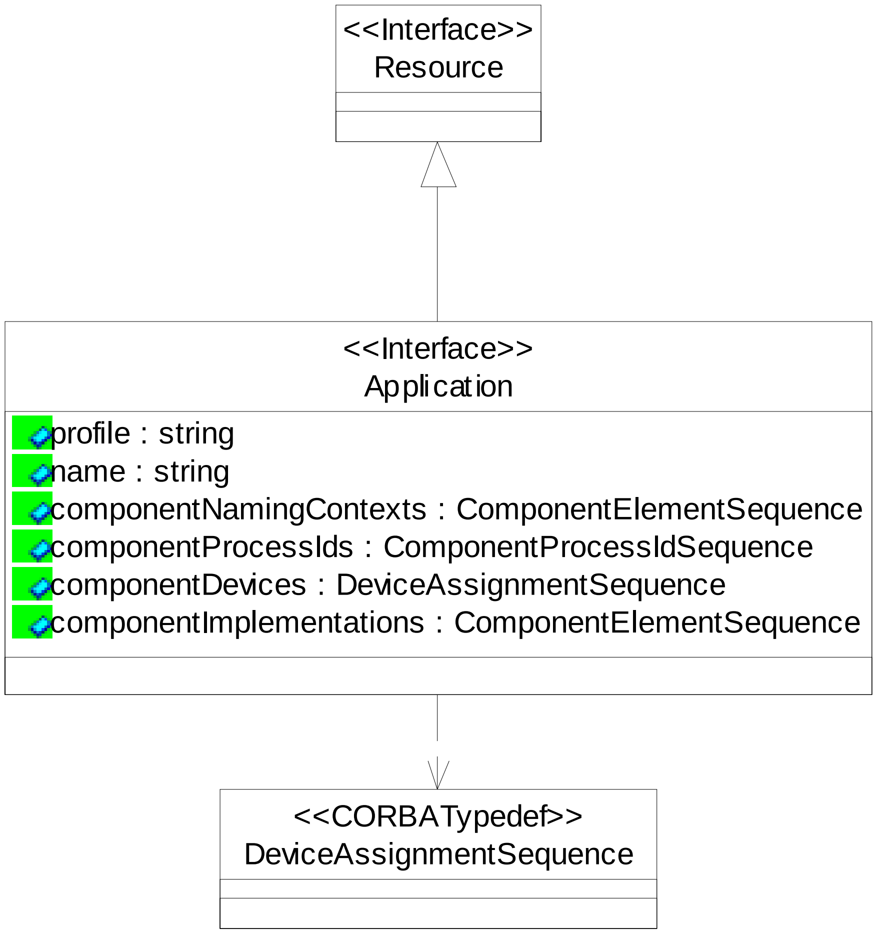

Figure 3-11. *Application* Interface UML

3.1.3.2.1.3 Types.

3.1.3.2.1.3.1 ComponentProcessIdType

The ComponetProcessIdType defines a type for associating a component with its process ID. This type can be used to retrieve a process ID for a specific component.

```
struct ComponentProcessIdType
{
      string componentID;
      unsigned long processID;
};
```

3.1.3.2.1.3.2 ComponentProcessIdSequence

The ComponetProcessIdSequence type defines an unbounded sequence of components' process IDs.

```
typedef sequence  <ComponentProcessIdType>  ComponentProcessIdSequence;
```

3.1.3.2.1.3.3 ComponentElementType

The ComponentElementType defines a type for associating a component with an element (e.g., naming context, implementation ID).

```
struct ComponentElementType
      {
      string componentID;
      string elementID;
      };
```

3.1.3.2.1.3.4 ComponentElementSequence

The componentElementSequence defines an unbounded sequence of components' with an associated element.

```
typedef sequence  <ComponentElementType>  ComponentElementSequence;
```

3.1.3.2.1.4 Attributes.

3.1.3.2.1.4.1 profile.

This profile attribute contains the Software Profile (3.1.3.4). CORBA-capable and non-CORBA-capable components have Profile files.

The readonly profile attribute shall contain either a profile element with a file reference to the SAD profile file or the XML for the SAD profile. Files referenced within a profile will have to be obtained from a *FileManager*. The *Application* will have to be queried for profile information for component files that are referenced by an ID instead of a file name.

```
readonly attribute string profile;
```

3.1.3.2.1.4.2 name.

This readonly name attribute shall contain the name of the created *Application*. The *ApplicationFactory* interface's *create* operation name parameter provides the name content.

```
readonly attribute string name;
```

3.1.3.2.1.4.3 componentNamingContexts.

The componentNamingContexts attribute shall contain the list of components' Naming Service Context within the Application for those components using CORBA Naming Service.

```
readonly attribute ComponentElementSequence componentNamingContexts;
```

3.1.3.2.1.4.4 componentProcessIds.

The componentProcessIds attribute shall contain the list of components' process IDs within the Application for components that are executing on a device.

```
readonly attribute ComponentProcessIdSequence componentProcessIds;
```

3.1.3.2.1.4.5 componentDevices.

The componentDevices attribute shall contain a list of components' device assignments within the application. Each component (componentinstantiation element in the Application's software profile) is associated with a device.

```
readonly attribute DeviceAssignmentSequence componentDevices;
```

3.1.3.2.1.4.6 componentImplementations.

The componentImplementations attribute shall contain the list of components' SPD implementation IDs within the Application for those components created.

```
readonly attribute ComponentElementSequence componentImplementations;
```

3.1.3.2.1.5 General Class Behavior.

The *Application* shall delegate the implementation of the inherited *Resource* operations (runTest, start, stop, configure, and query) to the application's assemblycontroller *Resource*. The *Application* shall propagate exceptions raised by the application's assemblycontroller's operations. The *initialize* operation shall not be propagated to *Application* components or the assemblycontroller, and causes no action within an *Application*.

3.1.3.2.1.6 Operations.

3.1.3.2.1.6.1 *releaseObject*.

3.1.3.2.1.6.1.1 Brief Rationale.

The *releaseObject* operation terminates execution of the *Application*, returns all allocated computing resources, and de-allocates the devices associated with *Application*. Before terminating, the *Application* removes the message connectivity with its associated *Applications* (e.g. *Ports*, *Resources*, and Logs) in the domain.

3.1.3.2.1.6.1.2 Synopsis.

```
void releaseObject() raises (ReleaseError);
```

3.1.3.2.1.6.1.3 Behavior.

The following behavior is in addition to the *LifeCycle releaseObject* operation behavior.

For each *Application* component not created by a *ResourceFactory,* the *releaseObject* operation shall release the component by utilizing the *Resources's releaseObject* operation. If the component was created by a *ResourceFactory*, the *releaseObject* operation shall release the component by the *ResourceFactory releaseResource* operation. The *releaseObject* operation shall shutdown a *ResourceFactory* when no more *Resources* are managed by the *ResourceFactory*. For each allocated device capable of operation execution, the *releaseObject* operation shall terminate all processes / tasks of the *Application's* components utilizing the *Device's terminate* operation.

For each allocated device capable of memory function, the *releaseObject* operation shall de-allocate the memory associated with *Application's* component instances utilizing the *Device's unload* operation.

The *releaseObject* operation shall deallocate the *Devices* that are associated with the *Application* being released, based on the *Application's* Software Profile. The actual devices deallocated (*Device deallocateCapacity*) will reflect changes in capacity based upon component capacity requirements deallocated from them, which may also cause state changes for the *Devices*.

The *Application* shall release all client component references to the *Application* components.

The *releaseObject* operation shall disconnect Ports from other Ports that have been connected based upon the software profile.

For components (e.g., *Resource, ResourceFactory*) that are registered with Naming Service, the *releaseObject* operation shall unregister those components from Naming Service.

The *releaseObject* operation for an application shall disconnect *Ports* first, then release the *Resources* and *ResourceFactories*, then call the terminate operation, and lastly call the unload operation on the devices.

The *releaseObject* operation shall, upon successful Application release, write an ADMINISTRATIVE_EVENT log record.

The *releaseObject* operation shall, upon unsuccessful Application release, write a FAILURE_ALARM log record.

The following steps demonstrate one scenario of the *Application*'s behavior for the release of an *Application* that contains *ResourceFactory* behavior:

1. Client invokes *releaseObject* operation.
2. Disconnects *Ports*.
3. Release the *ResourceFactory* components.
4. Shutdown the *ResourceFactory* components.
5. Release the *Resource* components.
6. Terminate the component's processes.
7. Unload the component's executable images.
8. Change the state of the associated device entries in the Domain Profile to be available, along with device(s) memory utilization availability and processor utilization availability based upon the Device Profile and Software Profile.
9. Unbind application components from Naming Service.
10. Log an Event indicating that the *Application* was either successfully or unsuccessfully released.
11. Remove the *Application* reference from the applications attribute.

Figure 3-12 is a collaboration diagram depicting the behavior as described above.

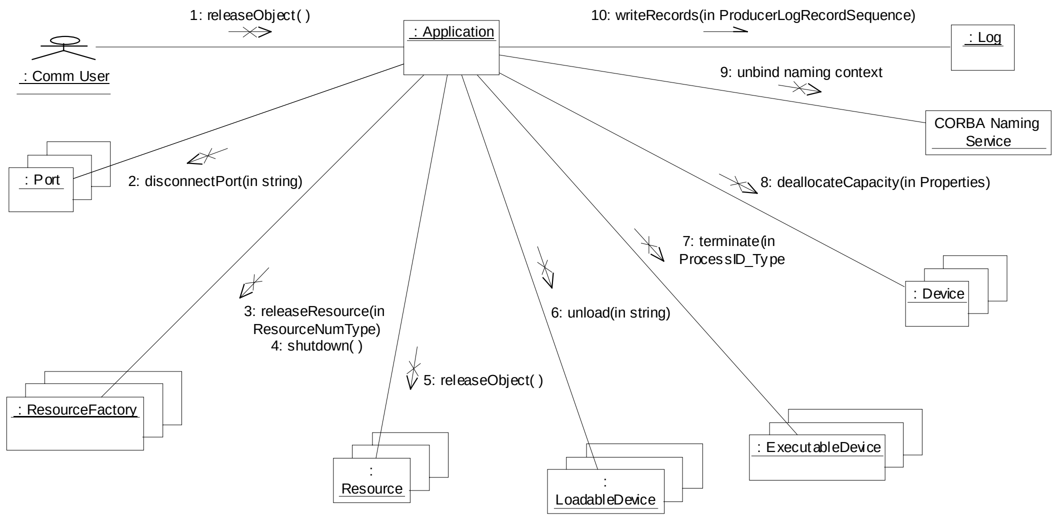

Figure 3-12. *Application* Behavior

3.1.3.2.1.6.1.4 Returns.

This operation does not return a value.

3.1.3.2.1.6.1.5 Exceptions/Errors.

The *releaseObject* operation shall raise a ReleaseError exception when the *releaseObject* operation unsuccessfully releases the *Application* components due to internal processing errors.

3.1.3.2.1.6.2 *getPort*.

3.1.3.2.1.6.2.1 Brief Rationale.

The *getPort* operation obtains a specific visible *Port* (e.g. command & control (HCI), data (red_io or black_io), responses of the *Application*.

3.1.3.2.1.6.2.2 Synopsis.

```
Object getPort(in string name) raises ( UnknownPort );
```

3.1.3.2.1.6.2.3 Behavior.

The *getPort* operation returns object references for port names that are in the *Application* SAD *externalports* element.

3.1.3.2.1.6.2.4 Returns.

The *getPort* operation shall return object references only for input port names that match the port names that are in the *Application* SAD externalports element.

3.1.3.2.1.6.2.5 Exceptions/Errors.

The *getPort* operation shall raise an UnknownPort exception if the port is invalid.

3.1.3.2.2 *ApplicationFactory*.

3.1.3.2.2.1 Description.

The *ApplicationFactory* interface class provides an interface to request the creation of a specific type of *Application* in the domain.

The *ApplicationFactory* interface class is designed using the Factory Design Pattern. The Software Profile determines the type of *Application* that is created by the *ApplicationFactory*.

3.1.3.2.2.2 UML.

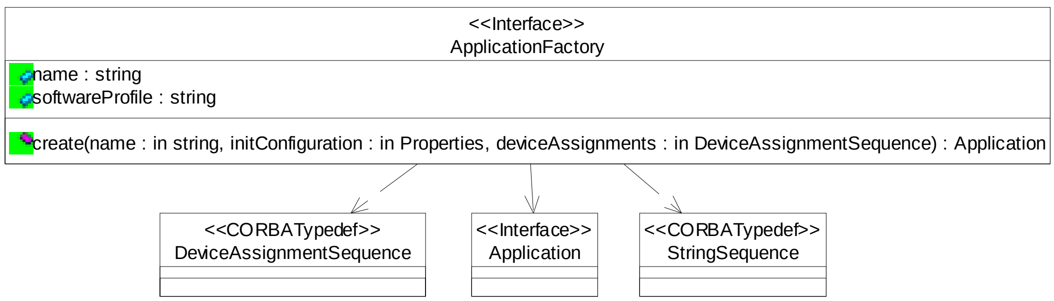

Figure 3-13. ***ApplicationFactory*** **UML**

3.1.3.2.2.3 Types.

3.1.3.2.2.3.1 CreateApplicationRequestError Exception.

The CreateApplicationRequestError exception is raised when the parameter CF DeviceAssignmentSequence contains one (1) or more invalid *Application* component-to-device assignment(s).

```
exception CreateApplicationRequestError
{
      DeviceAssignmentSequence      invalidAssignment;
}
```

3.1.3.2.2.3.2 CreateApplicationError Exception.

The CreateApplicationError exception is raised when a *create* request is valid but the *Application* is unsuccessfully instantiated due to internal processing errors.

```
exception CreateApplicationError
{
      StringSequence                errorMessages;
}
```

3.1.3.2.2.3.3 Exception InvalidInitConfiguration

The InvalidInitConfiguration exception is raised when the input initConfiguration parameter is invalid.

```
exception InvalidInitConfiguration
{
```

```
      Properties invalidProperties;
};
```

3.1.3.2.2.4 Attributes.

3.1.3.2.2.4.1 name.

The readonly name attribute shall contain the type of *Application* that can be instantiated by the *ApplicationFactory*.

```
readonly attribute string name;
```

3.1.3.2.2.4.2 softwareProfile.

The softwareProfile attribute contains the Software Profile for the *Application* that can be created by the *ApplicationFactory*.

The readonly softwareProfile attribute shall contain either a profile element with a file reference to the SAD profile or the XML for the SAD profile. Files referenced within the profile will have to be obtained from a *FileManager*. The *ApplicationFactory* will have to be queried for profile information for component files that are referenced by an ID instead of a file name.

```
readonly attribute string softwareProfile;
```

3.1.3.2.2.5 Operations.

3.1.3.2.2.5.1 *create*.

3.1.3.2.2.5.1.1 Brief Rationale.

The *create* operation is used to create an *Application* within the system domain.

The *create* operation provides a client interface to request the creation of an *Application* on client requested device(s) or the creation of an *Application* in which the *ApplicationFactory* determines the necessary device(s) required for instantiation of the *Application*.

3.1.3.2.2.5.1.2 Synopsis.

```
Application create(in string name, in Properties initConfiguration, in
DeviceAssignmentSequence deviceAssignments) raises ( CreateApplicationError,
CreateApplicationRequestError, InvalidInitConfiguration );
```

3.1.3.2.2.5.1.3 Behavior.

An *Application* can be comprised of one or more components (e.g., *Resources*, *Devices*, etc.). The SAD contains Software Package Descriptors (SPDs) for each *Application* component. The SPD specifies the *Device* implementation criteria for loading dependencies (processor kind, etc.) and processing capacities (e.g., memory, process) for an application component. The *create* operation shall use the SAD SPD implementation element to locate candidate devices capable of loading and executing *Application* components.

If deviceAssignments (not zero length) are provided, the *ApplicationFactory* verifies each device assignment, for the specified component, against the component’s SPD implementation element.

The *create* operation shall allocate (*Device allocateCapacity*) component capacity requirements against candidate devices to determine which candidate devices satisfy all SPD implementation criteria requirements and SAD partitioning requirements (e.g., components HostCollocation, etc.). The *create* operation shall only use *Devices* that have been granted successful capacity allocations for loading and executing *Application* components, or used for data processing. The actual *Devices* chosen will reflect changes in capacity based upon component capacity requirements allocated to them, which may also cause state changes for the *Devices*.

The *create* operation shall load the *Application* components (including all of the *Application*-dependent components) to the chosen device(s).

The *create* operation shall execute the application components (including all of the application-dependent components) as specified in the component's SPD implementation element's code element. The *create* operation shall use the component's SPD implementation code's stacksize and priority elements, when specified, for the execute options parameters. Parameters passed to entry points shall be in the form of a naming context parameter with an ID of "NAMING_CONTEXT" and string value of (/ DomainName / NodeName / [other context sequences]) / ComponentName_UniqueIdentifier, when the component object reference is to be retrieved from a Naming Service as indicated by the SAD. When the NAMING_CONTEXT parameter is used, the create operation shall form a naming service IOR with the format of an ID of "NAMING_SERVICE_IOR" and string value of the CORBA Naming Service IOR that the ApplicationFactory is using. The unique identifier is determined by the implementation, unique to each node. The *create* operation uses this naming string to form component names that need to be retrieved from Naming Service. (See also section 3.2.1.3.) Due to the dynamics of bind and resolve to Naming Service, the *create* operation should provide sufficient attempts to retrieve component object references from Naming Service prior to generating an exception.

In the naming parameter string, each "slash" (/) represents a separate naming context. A naming context is made up of ID and kind pair, which is indicated by "id.kind" in the naming parameter string. The naming context kind is optional and when not specified a null string will be used.

The NodeName naming context ID value is the label of the *DeviceManager* in which the component was loaded and going to be executed.

The ComponentName naming context ID value is the component instantiation findcomponent findby Naming Service name element value in the software assembly descriptor (SAD).

The *create* operation shall pass the componentinstantiation element "execparam" properties that have values as parameters to execute operation. The *create* operation passes "execparam" parameters values as string values.

The *create* operation shall, in order, initialize *Resources*, then establish connections for *Resources*, and finally configure the *Resources*.

The *create* operation will only configure the application's assemblycontroller component.

The *create* operation shall initialize an *Application* component provided the component implements the *LifeCycle* interface.

The create operation shall configure an application's assemblycontroller component provided the assemblycontroller has configure readwrite or writeonly properties with values. The *create* operation shall use the union of the input initConfiguration properties of the create operation and the assemblycontroller's componentinstantiation writeable "configure" properties that have values. The input initConfiguration parameter shall have precedence over the assemblycontroller's writeable "configure" property values. The *create* operation, when creating a component from a *ResourceFactory*, shall pass the componentinstantiation *componentresoursefactoryref* element “factoryparam” properties that have values as qualifiers parameters to the referenced *ResourceFactory* component’s *createResource* operation.

The *create* operation shall interconnect *Application* components' (*Resources'* or *Devices'*) ports in accordance with the SAD. It shall be possible to obtain *Ports* in accordance with the SAD via *PortSupplier getPort* operation. The *create* operation shall use the SAD *connectinterface* element id attribute as the unique identifier for a specific connection when provided. The *create* operation shall create a connection ID when no SAD connectinterface element attribute id is specified for a connection. The *create* operation shall obtain a *Resource* in accordance with the SAD via the CORBA Naming Service, *ResourceFactory*, or a stringified IOR. The*ResourceFactory* can be obtained by using the CORBA Naming Service or a stringified IOR as stated in the SAD.

The *create* operation shall pass, with invocation of each *ResourceFactory createResource* operation, the *ResourceFactory* configuration properties associated with that *Resource* as dictated by the SAD.

The dependencies to Log, *FileManager*, *FileSystem*, and CORBA Naming Service will be specified as connections in the SAD using the *domainfinder* element. The *create* operation will establish these connections. For connections established for a Log, the *create* operation shall create a unique producer log ID for each log producer. The *create* operation shall invoke the *PropertySet configure* operation once, and only once, per log producer (as described by the SAD *usesport* element) in order to set its unique PRODUCER_LOG_ID (see section 3.1.3.3.5.5.1.2 for details).

If the *Application* is successfully created, the *create* operation shall return an *Application* component reference for the created *Application*. A sequence of created *Application* references can be obtained using the *DomainManager getApplications* operation.

The *create* operation shall, upon successful Application creation, write an ADMINISTRATIVE_EVENT log record.

The *create* operation shall, upon unsuccessful Application creation, write a FAILURE_ALARM log record.

The dependencies to L*og*, *FileManager*, and *FileSystem* will appear as connections in the SAD using the *domainfinder* element. The *create* operation will establish these connections. For connections established for a Log, the *create* operation shall create a unique producer log ID one time for each log producer. The *create* operation shall invoke the *PropertySet configure* operation one time per log producer (as described by the SAD *usesport* element) in order to set its unique PRODUCER_LOG_ID (see section 3.1.3.3.5.5.1.2 for details).

The following steps demonstrate one scenario of the *ApplicationFactory’s* behavior for the creation of an *Application*:

1. Client invokes the *create* operation.
2. Evaluate the Domain Profile for available *Devices* that meet the *Application’s* memory and processor requirements, available Dependent Applications (e.g., *I/O* or *Utility* resources), and dependent libraries needed by the *Application*. Create an instance of an *Application*, if the requested *Application* can be created. Update the *Device*(s) memory and processor utilization.

3. Allocate the *Device*(s) memory and processor utilization.
4. Load the *Application* components on the devices using the appropriate *Device*(s) interface provided the *Application* component hasn't already been loaded.
5. Execute the *Application* components on the devices using the appropriate *Device* interface as indicated by the application's Software Profile.
6. Obtain the component reference (*Resource* or *ResourceFactory*) as described by the SAD.
7. If the component obtained from CORBA Naming Services is a *ResourceFactory* as indicated by the SAD, then narrow the component reference to be a *ResourceFactory* component.
8. If the component is a *ResourceFactory*, then create a *Resource* using the *ResourceFactory* interface.
9. If the components obtained from Naming Services are *Resources* supporting the *Resource* interface as indicated by the SCDs, then narrow the components reference to be *Resource* components.
10. Initialize the *Application*.
11. Get ports for the resources in order to interconnect the *Resources'* ports together.
12. Connect the ports that interconnect the *Resources'* ports together.
13. Configure the *Application*.
14. Return the *Application* object reference and log message.

Figure 3-14 is a collaboration diagram depicting the behavior as described above.

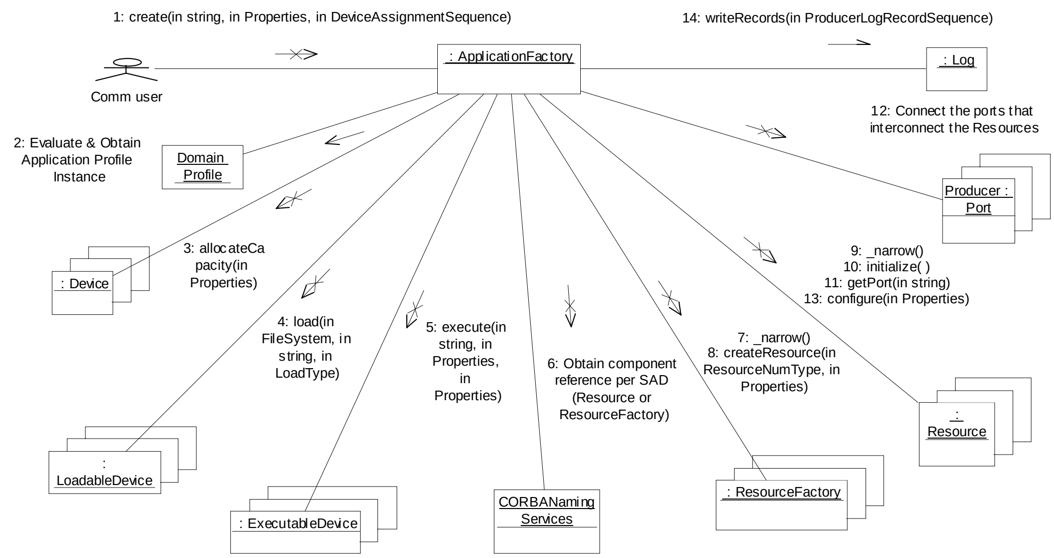

Figure 3-14. *ApplicationFactory* Behavior

3.1.3.2.2.5.1.4 Returns.

The *create* operation returns a duplicated *Application* reference for the created *Application*.

3.1.3.2.2.5.1.5 Exceptions/Errors.

The *create* operation shall raise the CreateApplicationRequestError exception when the parameter CF DeviceAssignmentSequence contains one (1) or more invalid *Application* component to device assignment(s).

The *create* operation shall raise the CreateApplicationError exception when the *create* request is valid but the *Application* can not be successfully instantiated due to internal processing error(s).

The *create* operation shall raise the InvalidInitConfiguration exception when the input initConfiguration parameter is invalid. The InvalidInitConfiguration invalidProperties shall identify the property that is invalid.

3.1.3.2.3 *DomainManager*.

3.1.3.2.3.1 Description.

The *DomainManager* interface is for the control and configuration of the system domain.

The *DomainManager* interface can be logically grouped into three categories: Human Computer Interface (HCI), Registration, and CF administration.

The HCI operations are used to configure the domain, get the domain's capabilities (*Devices*, Services, and *Applications*), and initiate maintenance functions. Host operations are performed by an HCI client capable of interfacing to the *DomainManager*.

The registration operations are used to register / unregister *DeviceManagers*, *DeviceManager's Devices*, *DeviceManager's* Services, and *Applications* at startup or during run-time for dynamic device, service, and application extraction and insertion.

The administration operations are used to access the interfaces of registered *DeviceManagers* and *DomainManager's FileManager*.

3.1.3.2.3.2 UML.

The DomainManager Interface UML is depicted in Figure 3-15.

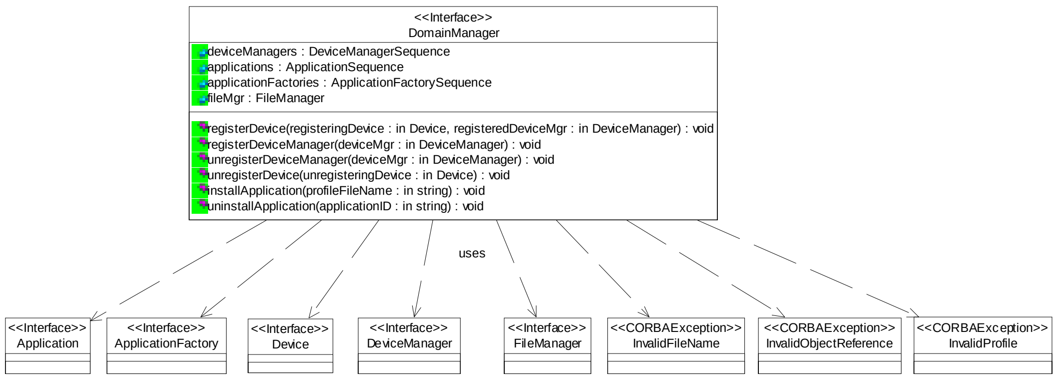

Figure 3-15. *DomainManager* Interface UML

3.1.3.2.3.3 Types.

3.1.3.2.3.3.1 exception ApplicationInstallationError.

The ApplicationInstallationError exception type is raised when an *Application* installation has not completed correctly.

```
exception  ApplicationInstallationError {};
```

3.1.3.2.3.3.2 InvalidIdentifier.

The InvalidIdentifier exception indicates an application identifier is invalid.

```
exception InvalidIdentifier {};
```

3.1.3.2.3.3.3 DeviceManagerSequence.

This type defines an unbounded sequence of *DeviceManager(s).*

```
typedef sequence <DeviceManager> DeviceManagerSequence
```

3.1.3.2.3.3.4 ApplicationSequence.

This type defines an unbounded sequence of *Application(s).*

```
typedef sequence < Application> ApplicationSequence
```

3.1.3.2.3.3.5 ApplicationFactorySequence.

This type defines an unbounded sequence of *ApplicationFactory(s).*

```
typedef sequence < ApplicationFactory> ApplicationFactorySequence
```

3.1.3.2.3.3.6 DeviceManagerNotRegistered Exception

The DeviceManagerNotRegistered exception indicates the registering *Device's DeviceManager* is not registered in the *DomainManager*. A *Device's DeviceManager* has to be registered prior to a *Device* registration to the *DomainManager*.

```
exception DeviceManagerNotRegistered {};
```

3.1.3.2.3.4 Attributes.

3.1.3.2.3.4.1 deviceManagers.

The deviceManagers attribute is read-only containing a sequence of registered *DeviceManagers* in the domain. The readonly *deviceManagers attribute* shall contain a list of registered DeviceManagers that have registered with the *DomainManager*. The *DomainManager* shall write an ADMINISTRATIVE_EVENT log to a *DomainManager's* Log, when the deviceManagers attribute is obtained by a client.

```
readonly attribute DeviceManagerSequence deviceManagers;
```

3.1.3.2.3.4.2 applications.

The applications attribute is read-only containing a sequence of instantiated *Applications* in the domain. The readonly applications attribute shall contain the list of *Applications* that have been instantiated. The *DomainManager* shall write an ADMINISTRATIVE_EVENT log record to a *DomainManager's* Log, when the application's attribute is obtained by a client.

```
readonly attribute ApplicationSequence applications;
```

3.1.3.2.3.4.3 applicationFactories.

The readonly applicationFactories attribute shall contain a list with one *ApplicationFactory* per application (SAD file and associated files) successfully installed (i.e. no exception raised). The *DomainManager* shall write an ADMINISTRATIVE_EVENT log record to a *DomainManager's* Log, when the applicationFactories attribute is obtained by a client.

```
readonly attribute ApplicationFactorySequence
      applicationFactories;
```

3.1.3.2.3.4.4 fileMgr.

The readonly fileMgr attribute shall contain the *DomainManager's FileManager*. The *DomainManager* shall write an ADMINISTRATIVE_EVENT log record to a *DomainManager's* Log, when the fileMgr attribute is obtained by a client.

```
readonly attribute FileManager fileMgr;
```

3.1.3.2.3.4.5 DomainManagerProfile.

The domainManagerProfile attribute contains the *DomainManager's* profile.

The readonly domainManagerProfile attribute shall contain either a profile element with a file reference to the *DomainManager* Configuration Descriptor (DMD) profile or the XML for the *DomainManager's* (DMD) profile. Files referenced within the profile will have to be obtained from the *DomainManager's FileManager*.

```
readonly attribute string domainManagerProfile;
```

3.1.3.2.3.5 General Class Behavior.

During component construction the *DomainManager* shall register itself with the CORBA Naming Service if available. During Naming Service registration the *DomainManager* shall create a naming context of "/DomainName" and bind the *DomainManager's* name with the created context. (If the CORBA Naming Service is not provided in an OE, DomainManager clients (e.g., *DeviceManager*) will have stringified IORs in their software profile entry in the Domain Profile.) (See also 3.1.3.2.2.5.1.3)

A *DomainManager* implementation may log to 0 to many Logs. The Logs utilized by the *DomainManager* implementation shall be defined in the DMD. See Appendix D for further description of the DMD file.

Once a service specified in the DMD is successfully registered with the *DomainManager* (via *registerDeviceManager* or *registerService* operations), the *DomainManager* shall begin to use the service (e.g. Log).

The *DomainManager* shall create its own *FileManager* component that consists of all registered *DeviceManager's FileSystems.*

The *DomainManager* shall restore *ApplicationFactories* after startup for applications that were previously installed by the *DomainManager installApplication* operation. The *DomainManager* shall add the restored *ApplicationFactories* to the *DomainManager's* applicationFactories attribute.

3.1.3.2.3.6 Operations.

3.1.3.2.3.6.1 *registerDeviceManager*.

3.1.3.2.3.6.1.1 Brief Rationale.

The *registerDeviceManager* operation is used to register a *DeviceManager*, its *Device*(s), and its Services. Software profiles can also be obtained from the *DeviceManager's FileSystem.*

3.1.3.2.3.6.1.2 Synopsis.

```
void registerDeviceManager(in DeviceManager deviceMgr) raises
(InvalidObjectReference, InvalidProfile );
```

3.1.3.2.3.6.1.3 Behavior.

The *registerDeviceManager* operation verifies that the input parameter, deviceMgr, is a not a nil CORBA component reference.

The *registerDeviceManager* operation shall add the input deviceMgr to the *DomainManager's* deviceManagers attribute, if it does not already exist. The *registerDeviceManager* operation shall add the input deviceMgr's registeredDevices and each registeredDevice's attributes (e.g., identifier, softwareProfile's allocation properties, etc.) to the *DomainManager*. The *registerDeviceManager* operation associates the input deviceMgr's with the input deviceMgr's registeredDevices in the *DomainManager* in order to support the *unregisterDeviceManager* operation.

The *registerDeviceManager* operation shall add the input deviceMgr's registeredServices and each registeredService's names to the *DomainManager*. The *registerDeviceManager* operation associates the input deviceMgr's with the input deviceMgr's registeredServices in the *DomainManager* in order to support the *unregisterDeviceManager operation*.

The *registerDeviceManager* operation shall perform the documented connections as specified in the *connections* element of the deviceMgr's DCD file. For connections established for a Log, the *registerDeviceManager* operation shall create a unique producer log ID for each log producer. The *registerDeviceManager* operation shall invoke the *PropertySet configure* operation once, and only once, on each log producer in order to set its unique PRODUCER_LOG_ID (see section 3.1.3.3.5.5.1.2 for details). If the *DeviceManager's* DCD describes a connection for a service that has not been registered with the *DomainManager*, the *registerDeviceManager* operation shall establish any pending connection when the service registers with the *DomainManager* by the *registerDeviceManager* operation.

The *registerDeviceManager* operation shall obtain all the Software profiles from the registering *DeviceManager's FileSystems*.

The *registerDeviceManager* operation shall mount the *DeviceManager's FileSystem* to the *DomainManager's* FileManager. The mounted FileSystem name shall have the format, "/DomainName/HostName", where DomainName is the name of the domain and HostName is the input deviceMgr's label attribute.

The *registerDeviceManager* operation shall, upon unsuccessful *DeviceManager* registration, write a FAILURE_ALARM log record to a *DomainManager's* Log.

The following UML sequence diagram (Figure 3-16) illustrates the *DomainManager's* behavior for the *registerDeviceManager* operation.

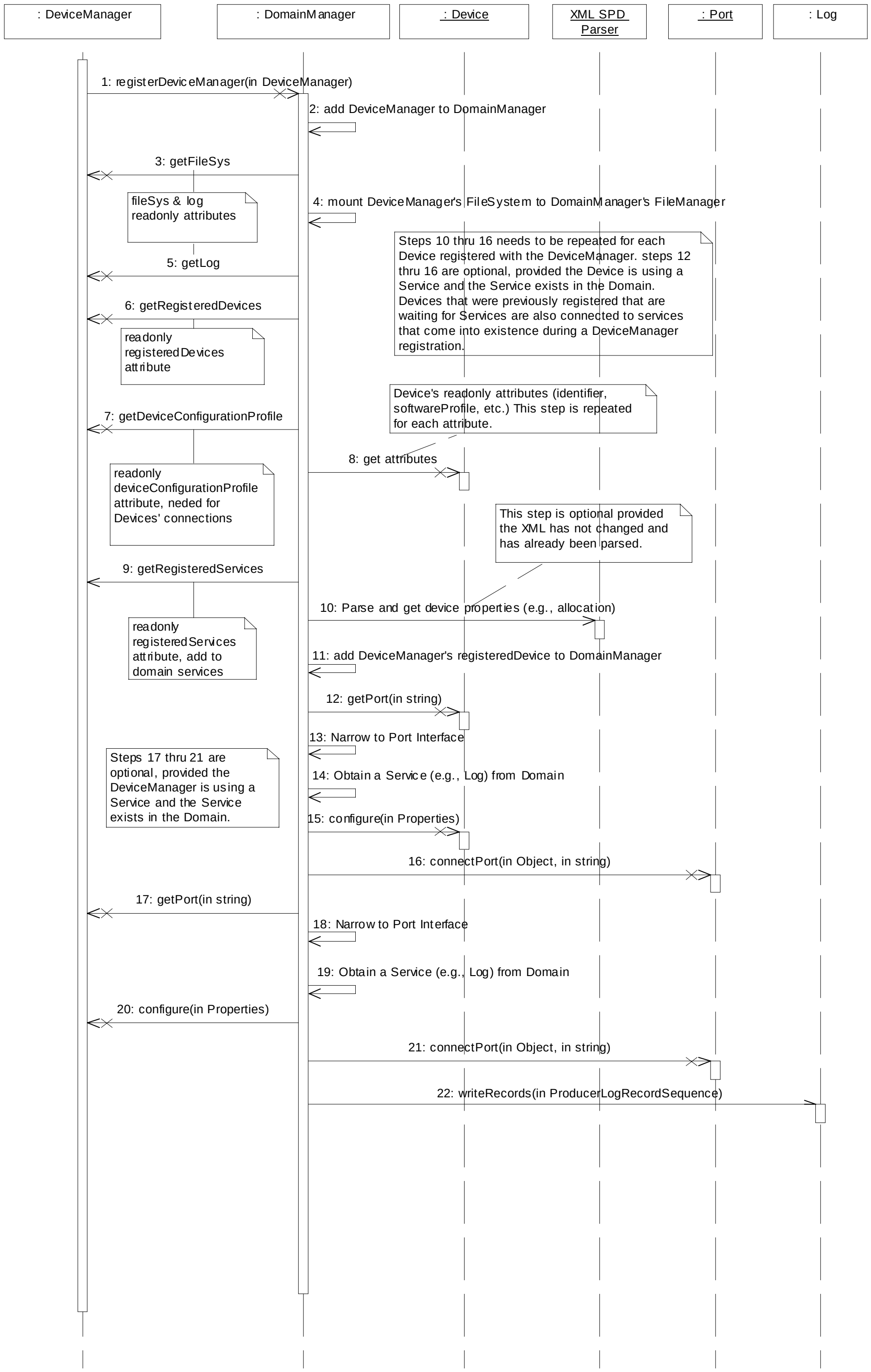

Figure 3-16. *DomainManager* Sequence Diagram for r*egisterDeviceManager* Operation

3.1.3.2.3.6.1.4 Returns.

This operation does not return a value.

3.1.3.2.3.6.1.5 Exceptions/Errors.

The *registerDeviceManager* operation shall raise the CF InvalidObjectReference exception when the input parameter *deviceMgr* contains an invalid reference to a *DeviceManager* interface.

3.1.3.2.3.6.2 *registerDevice*.

3.1.3.2.3.6.2.1 Brief Rationale.

The *registerDevice* operation is used to register a *Device* for a specific *DeviceManager* in the *DomainManager's* Domain Profile.

3.1.3.2.3.6.2.2 Synopsis.

```
void registerDevice(in Device registeringDevice, in DeviceManager
registeredDeviceMgr) raises (InvalidObjectReference, InvalidProfile,
DeviceManagerNotRegistered );
```

3.1.3.2.3.6.2.3 Behavior.

The *registerDevice* operation verifies that the input parameters, registeringDevice and registeredDeviceMgr, are not nil CORBA component references.

The *registerDevice* operation shall add the registeringDevice and the registeringDevice's attributes (e.g., identifier, softwareProfile's allocation properties, etc.) to the *DomainManager*, if it does not already exist.

The *registerDevice* operation associates the input registeringDevice with the input registeredDeviceMgr in the *DomainManager* when the input registeredDeviceMgr is a valid registered *DeviceManager* in the *DomainManager*.

The *registerDevice* operation shall, upon successful device registration, write an ADMINISTRATIVE_EVENT log record to a *DomainManager's* Log, to indicate that the device has successfully registered with the *DomainManager*.

Upon unsuccessful device registration, the *registerDevice* operation shall write a FAILURE_ALARM log record to a *DomainManager*'s Log, when the InvalidProfile exception is raised to indicate that the registeringDevice has an invalid profile.

Upon unsuccessful device registration, the *registerDevice* operation shall write a FAILURE_ALARM log record to a *DomainManager's* Log, indicating that the device could not register because the *DeviceManager* is not registered with the *DomainManager*.

Upon unsuccessful device registration, the *registerDevice* operation shall write a FAILURE_ALARM log record to a *DomainManager's* Log, because of an invalid reference input parameter.

The following UML sequence diagram (Figure 3-17) illustrates the *DomainManager's* behavior for the *registerDevice* operation.

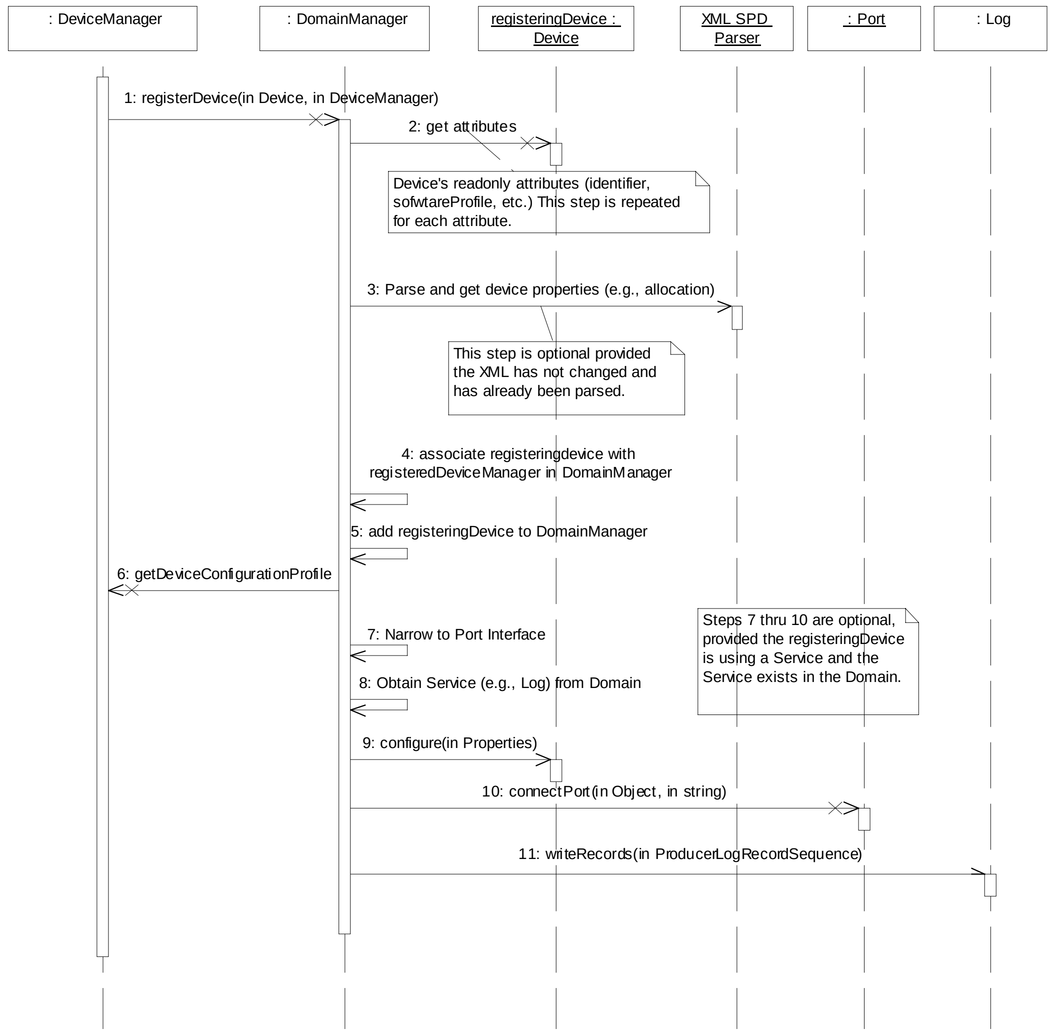

Figure 3-17. *DomainManager* Sequence Diagram for *registerDevice* Operation

3.1.3.2.3.6.2.4 Returns.

This operation does not return a value.

3.1.3.2.3.6.2.5 Exceptions/Errors.

The *registerDevice* operation shall raise the CF InvalidProfile exception when:

1. The *Device's* SPD file and the SPD's referenced files do not exist or cannot be processed due to the file not being compliant with XML syntax, or
2. The *Device's* SPD does not reference allocation properties.

The *registerDevice* operation shall raise a DeviceManagerNotRegistered exception when the input registeredDeviceMgr (not nil reference) is not registered with the *DomainManager*.

The *registerDevice* operation shall raise the CF InvalidObjectReference exception when input parameters registeringDevice or registeredDeviceMgr contains an invalid reference."

3.1.3.2.3.6.3 *installApplication*.

3.1.3.2.3.6.3.1 Brief Rationale.

The *installApplication* operation is used to install new application software in the *DomainManager's* Domain Profile. An installer application typically invokes this operation when it has completed the installation of a new application into the domain.

3.1.3.2.3.6.3.2 Synopsis.

```
void installApplication(in string profileFileName) raises ( InvalidProfile,
InvalidFileName, ApplicationInstallationError);
```

3.1.3.2.3.6.3.3 Behavior.

The profileFileName is the absolute path of the profile filename.

The *installApplication* operation shall verify the application's SAD file exists in the *DomainManager's* FileManager and all the files the application is dependent on are also resident.

The *installApplication* operation shall write an ADMINISTRATIVE_EVENT log record to a *DomainManager's* Log, upon successful *Application* installation.

The *installApplication* operation shall, upon unsuccessful application installation, write a FAILURE_ALARM log record to a *DomainManager's* Log.

3.1.3.2.3.6.3.4 Returns.

This operation does not return a value.

3.1.3.2.3.6.3.5 Exceptions/Errors.

The *installApplication* operation shall raise the ApplicationInstallationError exception when the installation of the *Application* file(s) was not successfully completed.

The *installApplication* operation shall raise the CF InvalidFileName exception when the input SAD file or any referenced file name does not exist in the file system as defined in the absolute path of the input profileFileName. When the CF InvalidFileName exception occurs, the installApplication operation shall log a FAILURE_ALARM log record to a *DomainManager's* Log with a message consisting of "installApplication::invalid file is xxx", where "xxx" is the input or referenced file name that is bad.

The *installApplication* operation shall raise the CF InvalidProfile exception when the input SAD file or any referenced file is not compliant with XML DTDs defined in Appendix D or referenced property definitions are missing. When the CF InvalidProfile exception occurs, the installApplication operation shall log a FAILURE_ALARM log record to a *DomainManager's* Log with a message consisting of "installApplication::invalid Profile is yyy," where "yyy" is the input or referenced file name that is bad along with the element or position within the profile that is bad.

3.1.3.2.3.6.4 *unregisterDeviceManager*.

3.1.3.2.3.6.4.1 Brief Rationale.

The *unregisterDeviceManager* operation is used to unregister a *DeviceManager* component from the *DomainManager's* Domain Profile. A *DeviceManager* may be unregistered during run-time for dynamic extraction or maintenance of the *DeviceManager*.

3.1.3.2.3.6.4.2 Synopsis.

```
void unregisterDeviceManager(in DeviceManager deviceMgr) raises
(InvalidObjectReference);
```

3.1.3.2.3.6.4.3 Behavior.

The *unregisterDeviceManager* operation shall unregister a *DeviceManager* component from the *DomainManager*.

The *unregisterDeviceManager* operation shall release all device(s) and service(s) associated with the *DeviceManager* that is being unregistered.

The *unregisterDeviceManager* operation shall unmount all *DeviceManager's FileSystems* from its *File Manager*.

The *unregisterDeviceManager* operation shall, upon the successful unregistration of a *DeviceManager*, write an ADMINISTRATIVE_EVENT log record to a *DomainManager's* Log.

The *unregisterDeviceManager* operation shall, upon unsuccessful unregistration of a *DeviceManager*, write a FAILURE_ALARM log record to a *DomainManager's* Log.

3.1.3.2.3.6.4.4 Returns.

This operation does not return a value.

3.1.3.2.3.6.4.5 Exceptions/Errors.

The *unregisterDeviceManager* operation shall raise the CF InvalidObjectReference when the input parameter *DeviceManager* contains an invalid reference to a *DeviceManager* interface.

3.1.3.2.3.6.5 *unregisterDevice*.

3.1.3.2.3.6.5.1 Brief Rationale.

The *unregisterDevice* operation is used to remove a device entry from the *DomainManager* for a specific *DeviceManager*.

3.1.3.2.3.6.5.2 Synopsis.

```
void unregisterDevice(in Device unregisteringDevice) raises
(InvalidObjectReference)
```

3.1.3.2.3.6.5.3 Behavior.

The *unregisterDevice* operation shall remove a device entry from the *DomainManager*.

The *unregisterDevice* operation shall release (client-side CORBA release) the unregisteringDevice from the Domain Manager.

The *unregisterDevice* operation shall, upon the successful unregistration of a *Device*, write an ADMINISTRATIVE_EVENT log record to a *DomainManager's* Log.

The *unregisterDevice* operation shall, upon unsuccessful unregistration of a *Device*, write a FAILURE_ALARM log record to a *DomainManager's Log*.

3.1.3.2.3.6.5.4 Returns.

This operation does not return a value.

3.1.3.2.3.6.5.5 Exceptions/Errors.

The *unregisterDevice* operation shall raise the CF InvalidObjectReference exception when the input parameter contains an invalid reference to a *Device* interface.

3.1.3.2.3.6.6 *uninstallApplication*.

3.1.3.2.3.6.6.1 Brief Rationale.

The *uninstallApplication* operation is used to uninstall an *ApplicationFactory* in the *DomainManager's* Domain Profile.

An installer application typically invokes this operation when removing an *ApplicationFactory* from the domain.

3.1.3.2.3.6.6.2 Synopsis.

```
void uninstallApplication(in String ApplicationID)raises (InvalidIdentifier);
```

3.1.3.2.3.6.6.3 Behavior.

The *uninstallApplication* operation shall remove all files associated with the *Application*.

The *uninstallApplication* operation shall make the *ApplicationFactory* unavailable from the *DomainManager* (i.e. its services no longer provided for the *Application*).

The *uninstallApplication* operation shall, upon successful uninstall of an *Application*, write an ADMINISTRATIVE_EVENT log record to a *DomainManager's* Log.

The *uninstallApplication* operation shall, upon unsuccessful uninstall of an *Application,* write a FAILURE_ALARM log record to a *DomainManager's* Log.

The *uninstallApplication* operation shall, upon unsuccessful uninstall of an *Application*, log a FAILURE_ALARM log record to a *DomainManager's* Log.

3.1.3.2.3.6.6.4 Returns.

This operation does not return a value.

3.1.3.2.3.6.6.5 Exceptions/Errors.

The *uninstallApplication* operation shall raise the InvalidIdentifier exception when the ApplicationID is invalid.

3.1.3.2.3.6.7 *registerService*.

3.1.3.2.3.6.7.1 Brief Rationale.

The *registerService* operation is used to register a service for a specific *DeviceManager* with the *DomainManager*.

3.1.3.2.3.6.7.2 Synopsis.

```
void registerService(in Object registeringService, in DeviceManager
registeredDeviceMgr, in string name) raises (InvalidObjectReference,
DeviceManagerNotRegistered);
```

3.1.3.2.3.6.7.3 Behavior.

The *registerService* operation shall verify the input registeringService and registeredDeviceMgr are valid object references.

The *registerService* operation shall verify the input registeredDeviceMgr has been previously registered with the *DomainManager*.

The *registerService* operation shall add the registeringService and the registeringService's name to the *DomainManager* if the name of the registering service is a unique name. However, if the name of the registering service is not a unique name for that type of service (i.e. it is a duplicate name of an already registered service of the same type of service), then the new service shall not be registered by the *DomainManager*.

The *registerService* operation shall associate the input registeringService with the input registeredDeviceMgr in the *DomainManager* when the input registeredDeviceMgr is a valid registered *DeviceManager* in the *DomainManager*.

The *registerService* operation shall, upon successful service registration, establish any pending connection requests for the registeringService. For connections established for a *Log*, the *registerService* operation shall create a unique producer log ID for each log producer. The registerService operation shall invoke the *PropertySet configure* operation once, and only once, on each log producer in order to set its unique PRODUCER_LOG_ID (see section 3.1.3.3.5.5.1.2 for *details*).

The *registerService* operation shall, upon successful service registration, write an ADMINISTRATIVE_EVENT log record to a *DomainManager's* Log.

The *registerService* operation shall, upon unsuccessful service registration, write a FAILURE_ALARM log record to a *DomainManager's* Log.

The following UML sequence diagram (Figure 3-18) illustrates the *DomainManager's* behavior for the *registerService* operation.

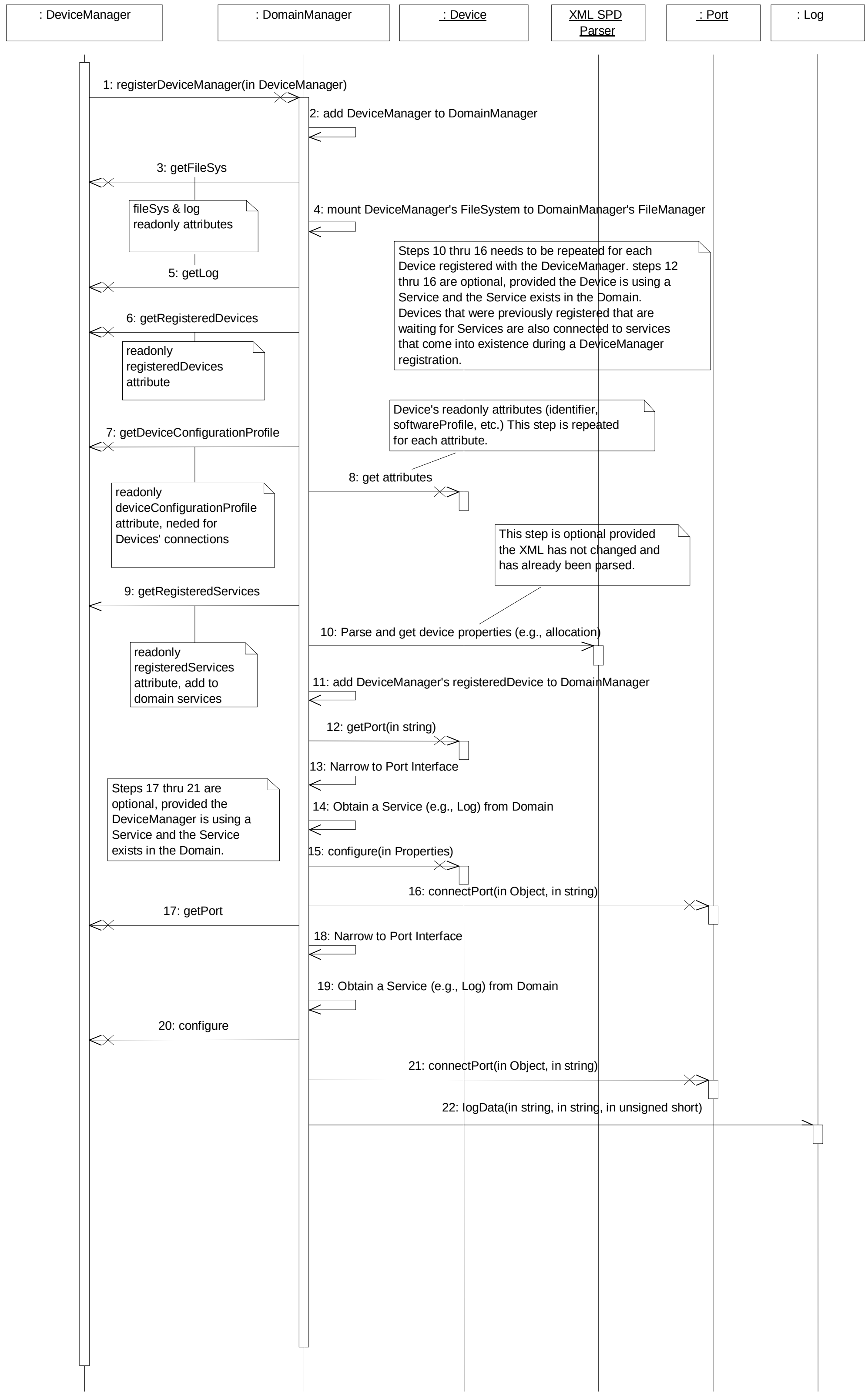

Figure 3-18. ***DomainManager*** **Sequence Diagram for** ***registerService*** **Operation**

3.1.3.2.3.6.7.4 Returns.

This operation does not return a value.

3.1.3.2.3.6.7.5 Exceptions/Errors.

The *registerService* operation shall raise a DeviceManagerNotRegistered exception when the input registeredDeviceMgr (not nil reference) is not registered with the *DomainManager*.

The *registerService* operation shall raise the CF InvalidObjectReference exception when input parameters registeringService or registeredDeviceMgr contains an invalid reference.

3.1.3.2.3.6.8 *unregisterService*.

3.1.3.2.3.6.8.1 Brief Rationale.

The *unregisterService* operation is used to remove a service entry from the *DomainManager* for a specific *DeviceManager*.

3.1.3.2.3.6.8.2 Synopsis.

```
void unregisterService(in Object unregisteringService, in string name) raises
( InvalidObjectReference);
```

3.1.3.2.3.6.8.3 Behavior.

The *unregisterService* operation shall remove a service entry from the *DomainManager*.

The *unregisterService* operation shall release (client-side CORBA release) the unregisteringService from the *DomainManager*.

The *unregisterService* operation shall, upon the successful unregistration of a Service, write an ADMINISTRATIVE_EVENT log record to a *DomainManager's* Log.

The *unregisterService* operation shall, upon unsuccessful unregistration of a Service, write a FAILURE_ALARM log record to a *DomainManager's* Log.

3.1.3.2.3.6.8.4 Returns.

This operation does not return a value.

3.1.3.2.3.6.8.5 Exceptions/Errors.

The *unregisterService* operation shall raise the CF InvalidObjectReference exception when the input parameter contains an invalid reference to a Service interface.

3.1.3.2.4 *Device*.

3.1.3.2.4.1 Description.

A *Device* is a type of *Resource* within the domain and has the requirements as stated in the *Resource* interface. This interface defines additional capabilities and attributes for any logical *Device* in the domain. A logical *Device* is a functional abstraction for a set (e.g., zero or more) of hardware devices and provides the following attributes and operations:

1. Software Profile Attribute – This SPD XML profile defines the logical *Device* capabilities (data/command uses and provides ports, configure and query properties, capacity properties, status properties, etc.), which could be a subset of the hardware device's capabilities.
2. State Management & Status Attributes – This information describes the administrative, usage, and operational states of the device.

3. Capacity Operations **-** In order to use a device, certain capacities (e.g., memory, performance, etc.) must be obtained from the *Device*. The capacity properties will vary among devices and are described in the Software Profile. A device may have multiple allocatable capacities, each having its own unique capacity model.

3.1.3.2.4.2 UML.

The *Device* Interface UML is depicted in figure 3-18.

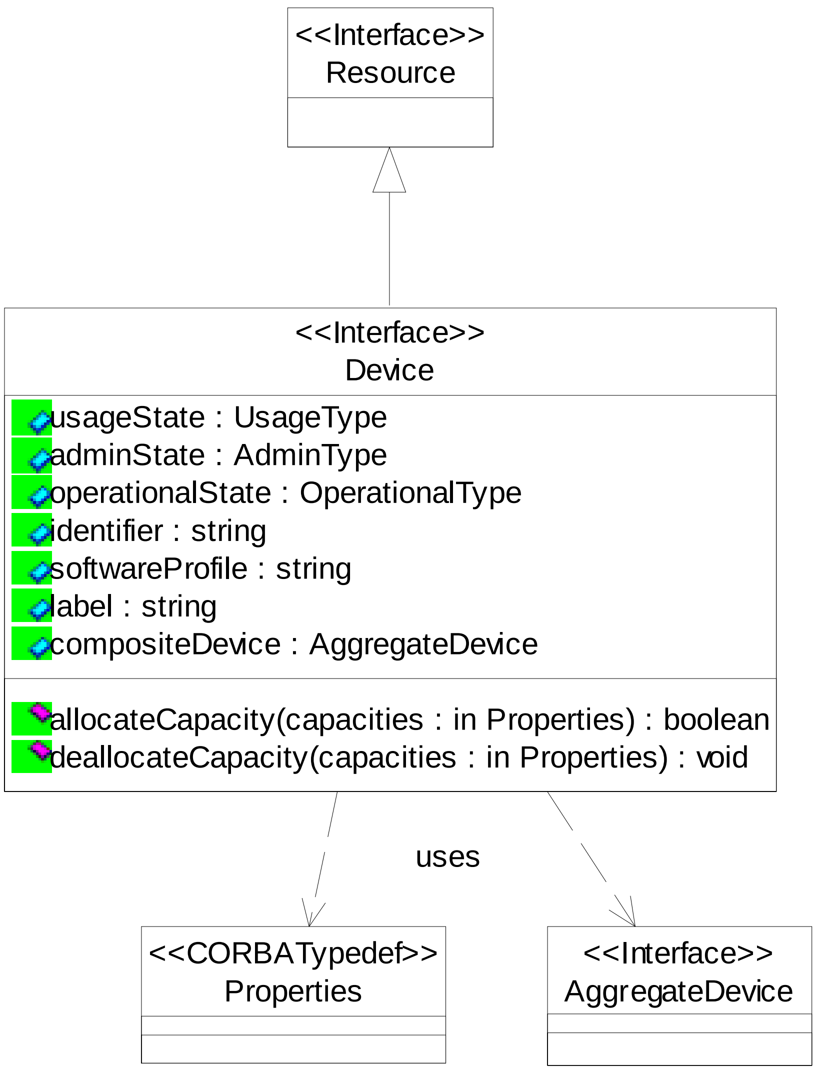

Figure 3-19. *Device* Interface UML

3.1.3.2.4.3 Types.

3.1.3.2.4.3.1 InvalidState.

The InvalidState exception indicates that the device is not capable of the behavior being attempted due to the state the *Device* is in. An example of such behavior is *allocateCapacity*.

```
exception DeviceNotCapable InvalidState {string msg;};
```

3.1.3.2.4.3.2 InvalidCapacity.

The InvalidCapacity exception returns the capacities that are not valid for this device.

```
exception InvalidCapacity
{
      string      msg;
      Properties  capacities;
};
```

3.1.3.2.4.3.3 AdminType.

This is a CORBA IDL enumeration type that defines a *Device's* administrative states. The administrative state indicates the permission to use or prohibition against using the *Device*.

```
enum AdminType
    {
      LOCKED,
      SHUTTING_DOWN,
      UNLOCKED
    };
```

3.1.3.2.4.3.4 OperationalType.

This is a CORBA IDL enumeration type that defines a *Device's* operational states. The operational state indicates whether or not the object is functioning.

```
enum OperationalType
    {
      ENABLED,
      DISABLED
    };
```

3.1.3.2.4.3.5 UsageType.

This is a CORBA IDL enumeration type that defines the *Devices* usage states. This state indicates whether or not a *Device* is:

- IDLE – not in use
- ACTIVE – in use, with capacity remaining for allocation, or
- BUSY – in use, with no capacity remaining for allocation

```
enum UsageType
    {
      IDLE,
      ACTIVE,
      BUSY
    };
```

3.1.3.2.4.4 Attributes.

3.1.3.2.4.4.1 usageState.

The readonly usageState attribute shall contain the *Device's* usage state (IDLE, ACTIVE, or BUSY, see Figure 3-21). UsageState indicates whether or not a device is actively in use at a specific instant, and if so, whether or not it has spare capacity for allocation at that instant.

```
readonly attribute UsageType usageState;
```

3.1.3.2.4.4.2 adminState.

The administrative state indicates the permission to use or prohibition against using the device. The adminState attribute shall contain the device's admin state value. The adminState attribute shall only allow the setting of LOCKED and UNLOCKED values, where "LOCKED" is only effective when the adminState attribute value is UNLOCKED, and "UNLOCKED" is only effective when the adminState attribute value is LOCKED or SHUTTING_DOWN. Illegal state transitions commands are ignored.

The adminState attribute, upon being commanded to be LOCKED, shall transition from the UNLOCKED to the SHUTTING_DOWN state and set the adminState to LOCKED for its entire aggregation of *Devices* (if it has any). The adminState shall then transition to the LOCKED state when the *Device's* usageState is IDLE and its entire aggregation of *Devices* are LOCKED. Refer to Figure 3-19 for an illustration of the above state behavior.

```
attribute AdminType adminState;
```

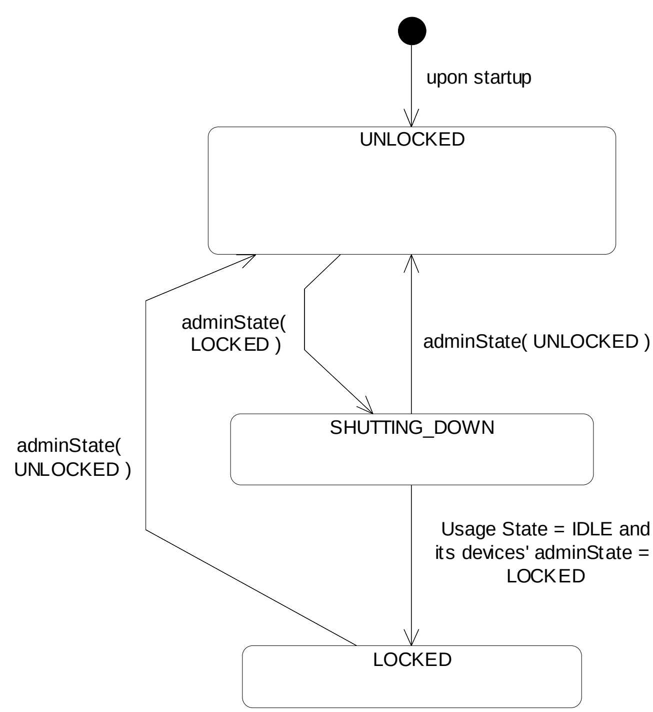

Figure 3-20. State Transition Diagram for adminState

3.1.3.2.4.4.3 operationalState.

The readonly operationalState attribute shall contain the device's operational state (ENABLED or DISABLED). The operational state indicates whether or not the device is functioning.

```
readonly attribute OperationalType operationalState;
```

3.1.3.2.4.4.4 identifier.

The readonly identifier attribute shall contain the unique identifier for a device instance.

```
readonly attribute string identifier;
```

3.1.3.2.4.4.5 softwareProfile.

The softwareProfile attribute is the XML software description for this logical *Device*.

The readonly softwareProfile attribute shall contain either a profile DTD element with a file reference to the SPD profile file or the XML for the SPD profile. Files referenced within the softwareProfile are obtained via the *FileManager*.

```
readonly attribute string softwareProfile;
```

3.1.3.2.4.4.6 label.

The readonly label attribute shall contain the *Device's* label. The label attribute is the meaningful name given to a *Device*. The attribute could convey location information within the system (e.g., audio1, serial1, etc.).

```
readonly attribute string label;
```

3.1.3.2.4.4.7 compositeDevice.

The readonly compositeDevice attribute shall contain the object reference of the *aggregateDevice* which this *Device* is associated with or a nil CORBA object reference if no association exists.

```
readonly attribute AggregateDevice compositeDevice;
```

3.1.3.2.4.5 Operations.

3.1.3.2.4.5.1 *allocateCapacity*.

3.1.3.2.4.5.1.1 Brief Rationale.

The *allocateCapacity* operation provides the mechanism to request and allocate capacity from the *Device*.

3.1.3.2.4.5.1.2 Synopsis.

```
boolean allocateCapacity(in Properties capacities) raises ( InvalidCapacity,
InvalidState );
```

3.1.3.2.4.5.1.3 Behavior.

The *allocateCapacity* operation shall reduce the current capacities of the *Device* based upon the capacities requested, when the adminState is UNLOCKED, operationalState is ENABLED, and usageState is not BUSY.

The *allocateCapacity* operation shall set the *Device's* usageState attribute to BUSY, when the *Device* determines that it is not possible to allocate any further capacity. The *allocateCapacity* operation shall set the usageState attribute to ACTIVE, when capacity is being used and any capacity is still available for allocation (reference Figure 3-21).

3.1.3.2.4.5.1.4 Returns.

The *allocateCapacity* operation shall return true, if the capacity has been allocated, or false if not allocated.

3.1.3.2.4.5.1.5 Exceptions/Errors.

The *allocateCapacity* operation shall raise the InvalidCapacity exception, when the capacities are invalid or the capacity values are the wrong type or ID.

The *allocateCapacity* operation shall raise the InvalidState exception, when the *Device's* adminState is not UNLOCKED or operationalState is DISABLED.

3.1.3.2.4.5.2 *deallocateCapacity*.

3.1.3.2.4.5.2.1 Brief Rationale.

The *deallocateCapacity* operation provides the mechanism to return capacities back to the *Device*, making them available to other users.

3.1.3.2.4.5.2.2 Synopsis.

```
void deallocateCapacity(in Properties capacities) raises ( InvalidCapacity,
InvalidState );
```

3.1.3.2.4.5.2.3 Behavior.

The *deallocateCapacity* operation shall adjust the current capacities of the *Device* based *upon* the input capacities parameter.

The *deallocateCapacity* operation shall set the usageState attribute to ACTIVE when, after adjusting capacities, any of the *Device's* capacities are still being used.

The *deallocateCapacity* operation shall set the usageState attribute to IDLE when, after adjusting capacities, none of the *Device's* capacities are still being used.

The *deallocateCapacity* operation shall set the adminState attribute to LOCKED as specified in 3.1.3.2.4.4.2.

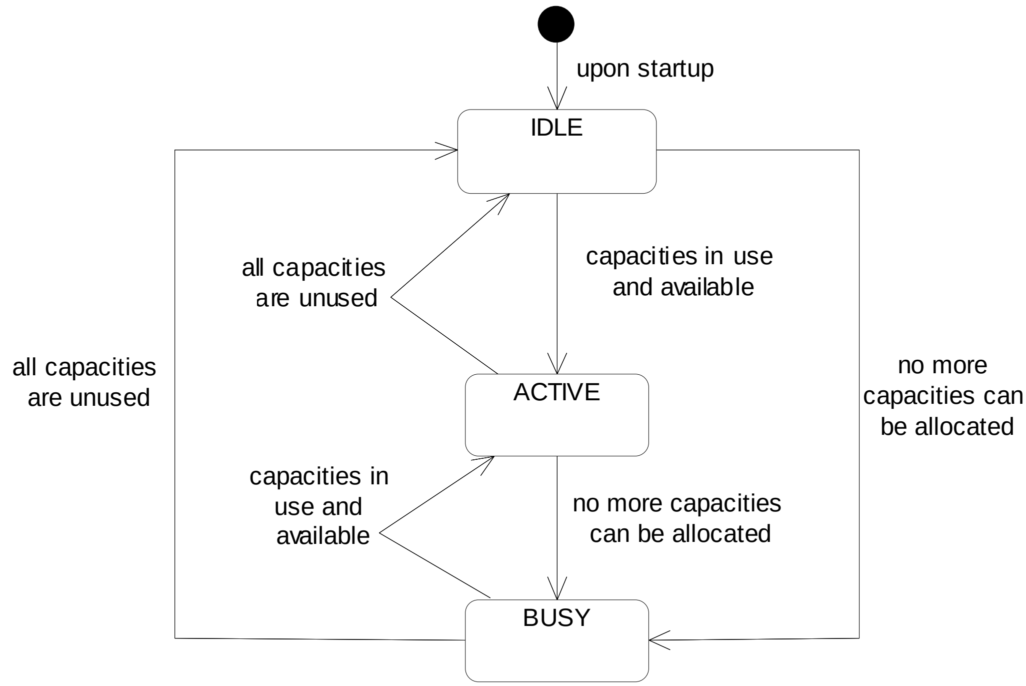

Figure 3-21. State Transition Diagram for *allocateCapacity* and *deallocateCapacity*

3.1.3.2.4.5.2.4 Returns.

This operation does not return any value.

3.1.3.2.4.5.2.5 Exceptions/Errors.

The *deallocateCapacity* operation shall raise the InvalidCapacity exception, when the capacity ID is invalid or the capacity value is the wrong type. The InvalidCapacity exception will state the reason for the exception.

The *deallocateCapacity* operation shall raise the InvalidState exception, when the *Device's* adminState is LOCKED or operationalState is DISABLED.

3.1.3.2.4.5.3 *releaseObject*.

3.1.3.2.4.5.3.1 Description.

This section describes additional release behavior for a logical *Device*.

3.1.3.2.4.5.3.2 Synopsis.

```
void releaseObject() raises (ReleaseError);
```

3.1.3.2.4.5.3.3 Behavior.

The following behavior is in addition to the *LifeCycle releaseObject* operation behavior.

The *releaseObject* operation shall call the *releaseObject* operation on all of the *Device's* aggregated *Devices* (i.e., those *Devices* that are contained within the *AggregateDevice's* devices attribute).

The *releaseObject* operation shall transition the *Device's* adminState to SHUTTING_DOWN state, when the *Device's* adminState is UNLOCKED.

The *releaseObject* operation shall cause the *Device* to be unavailable (i.e. released from the CORBA environment, and process terminated on the OS when applicable), when the *Device's* adminState transitions to LOCKED, meaning its aggregated *Devices* have been removed and the *Device's* usageState is IDLE.

The *releaseObject* operation shall cause the removal of its *Device* from the *Device's* compositeDevice.

The *releaseObject* operation shall unregister its *Device* from its *DeviceManager*.

The following four figures (3-22, 3-23, 3-24, and 3-25) depict different release scenarios depending on the type of *Device* and the state the *Device* is in.

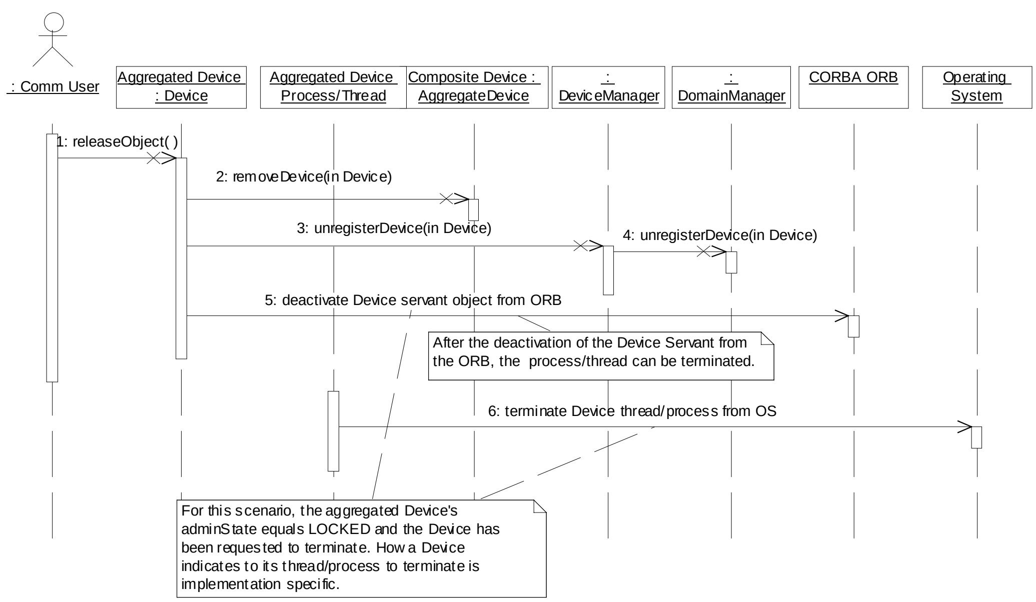

Figure 3-22. Release Aggregated *Device* Scenario

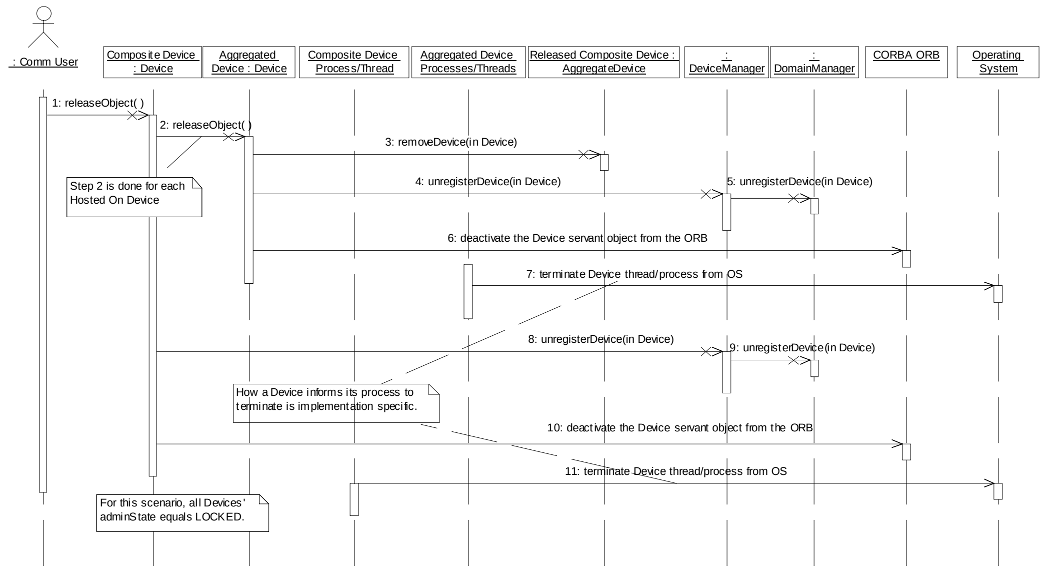

Figure 3-23. Release Composite *Device* Scenario

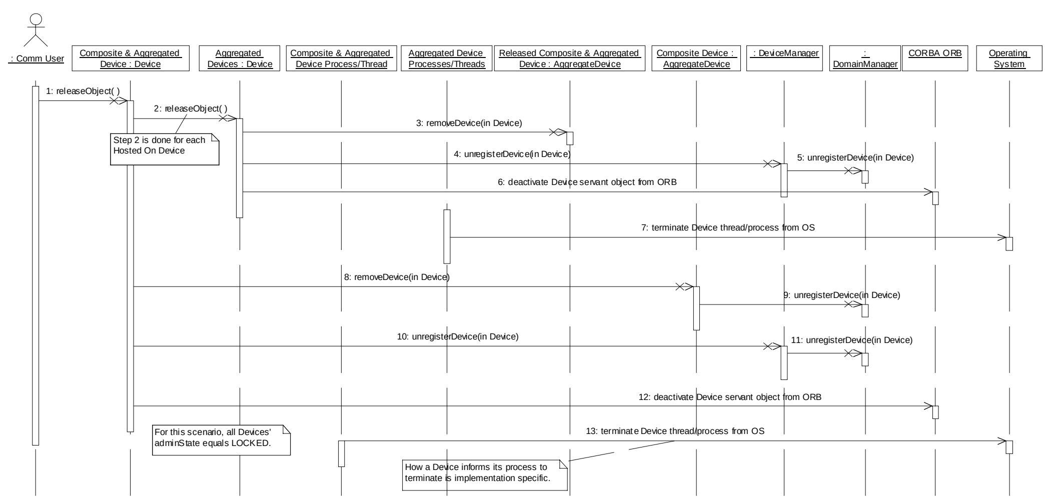

Figure 3-24. Release Composite & Aggregated *Device* Scenario

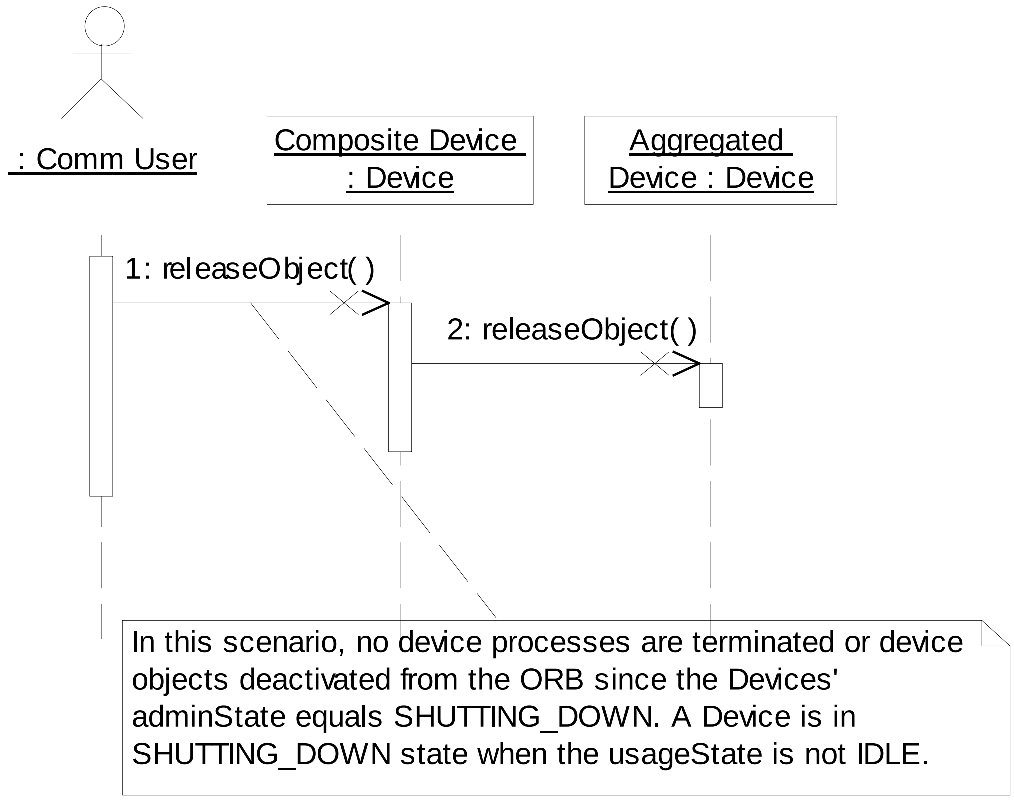

Figure 3-25. Release Composite *Device* in SHUTTING_DOWN State Scenario

3.1.3.2.4.5.3.4 Returns.

The *releaseObject* operation does not return a value.

3.1.3.2.4.5.3.5 Exceptions/Errors.

The *releaseObject* operation shall raise the ReleaseError exception when the *releaseObject* is not successful in releasing a logical *Device* due to internal processing errors that occurred within the *Device* being *released*.

3.1.3.2.5 *LoadableDevice*.

3.1.3.2.5.1 Description.

This interface extends the *Device* interface by adding software loading and unloading behavior to a *Device*.

3.1.3.2.5.2 UML.

The *LoadableDevice* Interface UML is depicted in Figure 3-26 *below*.

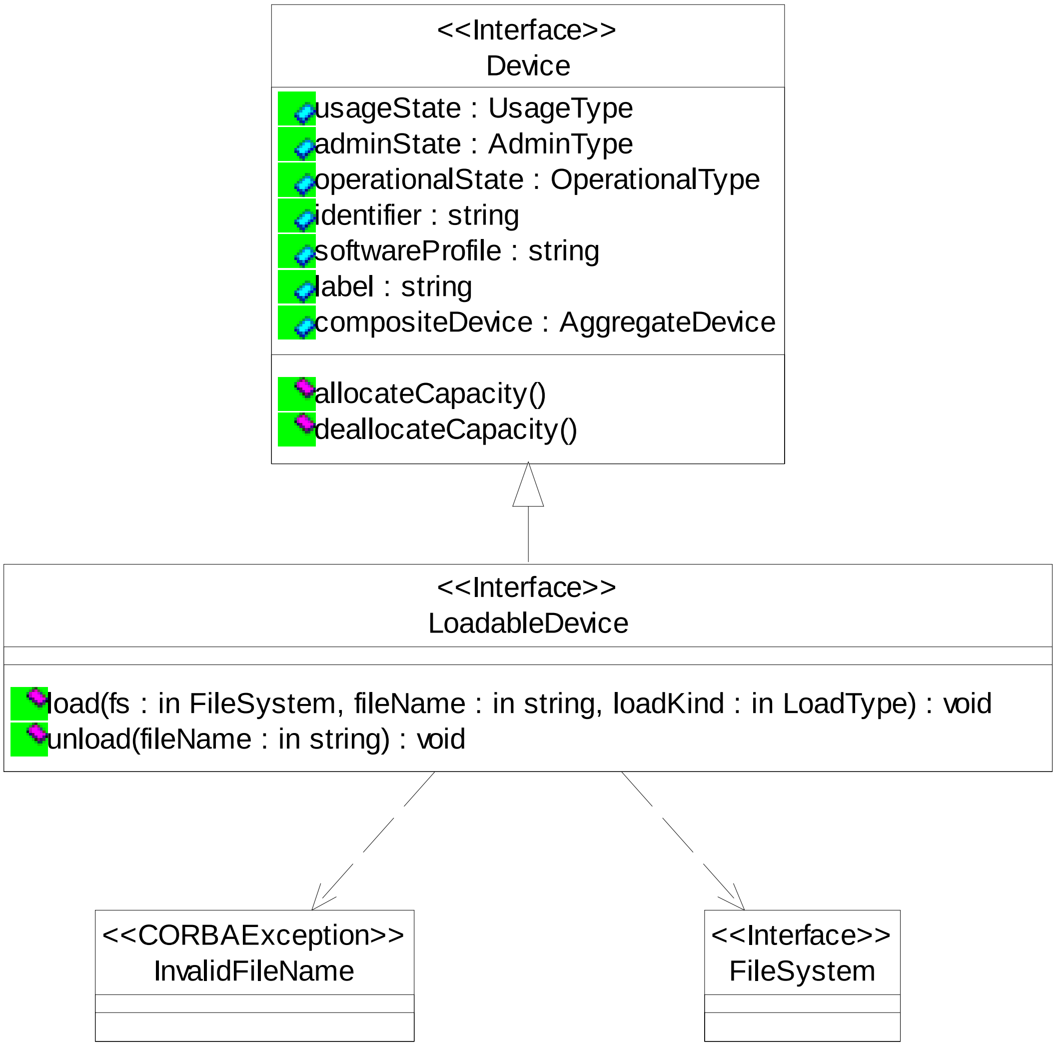

Figure 3-26. *LoadableDevice* Interface UML

3.1.3.2.5.3 Types.

3.1.3.2.5.3.1 LoadType.

This type defines the type of load to be performed. The load types are in accordance with Appendix D.

```
enum LoadType
    {
      KERNEL_MODULE,
      DRIVER,
      SHARED_LIBRARY,
      EXECUTABLE
  };
```

3.1.3.2.5.3.2 InvalidLoadKind.

The InvalidLoadKind exception indicates that the device is unable to load that type of file, as identified by the loadKind parameter.

```
exception InvalidLoadKind {};
```

3.1.3.2.5.4 Attributes.

N/A

3.1.3.2.5.5 Operations.

3.1.3.2.5.5.1 *load*.

3.1.3.2.5.5.1.1 Brief Rationale.

The *load* operation provides the mechanism for loading software on a specific device. The loaded software may be subsequently executed on the *Device*, if the *Device* is an *ExecutableDevice*.

3.1.3.2.5.5.1.2 Synopsis.

```
void load(in FileSystem fs, in string fileName, in LoadType loadKind)
  raises (InvalidState, InvalidLoadKind, InvalidFileName );
```

3.1.3.2.5.5.1.3 Behavior.

The *load* operation shall load a file on the specified device based upon the input loadKind and fileName parameters using the input FileSystem parameter to retrieve the file.

The *load* operation shall support the load types as stated in the Device's software profile LoadType allocation properties.

The *load* operation shall keep track of the number of times a file has been successfully loaded.

3.1.3.2.5.5.1.4 Returns.

This operation does not return any value.

3.1.3.2.5.5.1.5 Exceptions/Errors.

The *load* operation shall raise the InvalidState exception when the *Device's* adminState is not UNLOCKED or operationalState is DISABLED.

The load operation shall raise the InvalidLoadKind exception when the input loadKind parameter is not supported.

The load operation shall raise the CF InvalidFileName exception when the file designated by the input filename parameter cannot be found.

3.1.3.2.5.5.2 *unload*.

3.1.3.2.5.5.2.1 Brief Rationale.

The *unload* operation provides the mechanism to unload software that is currently loaded.

3.1.3.2.5.5.2.2 Synopsis.

```
void unload(in string fileName) raises ( InvalidState, InvalidFileName );
```

3.1.3.2.5.5.2.3 Behavior.

The *unload* operation shall decrement the load count for the input filename parameter by one. The *unload* operation shall unload the application software on the device based on the input fileName parameter, when the file's load count equals zero.

3.1.3.2.5.5.2.4 Returns.

This operation does not return a value.

3.1.3.2.5.5.2.5 Exceptions/Errors.

The *unload* operation shall raise the InvalidState exception when the *Device's* adminState is LOCKED or its operationalState is DISABLED.

The *unload* operation shall raise the CF InvalidFileName exception when the file designated by the input filename parameter cannot be found.

3.1.3.2.6 *ExecutableDevice*.

3.1.3.2.6.1 Description.

This interface extends the *LoadableDevice* interface by adding execute and terminate behavior to a *Device*.

3.1.3.2.6.2 UML.

The *ExecutableDevice* Interface UML is depicted in Figure 3-27.

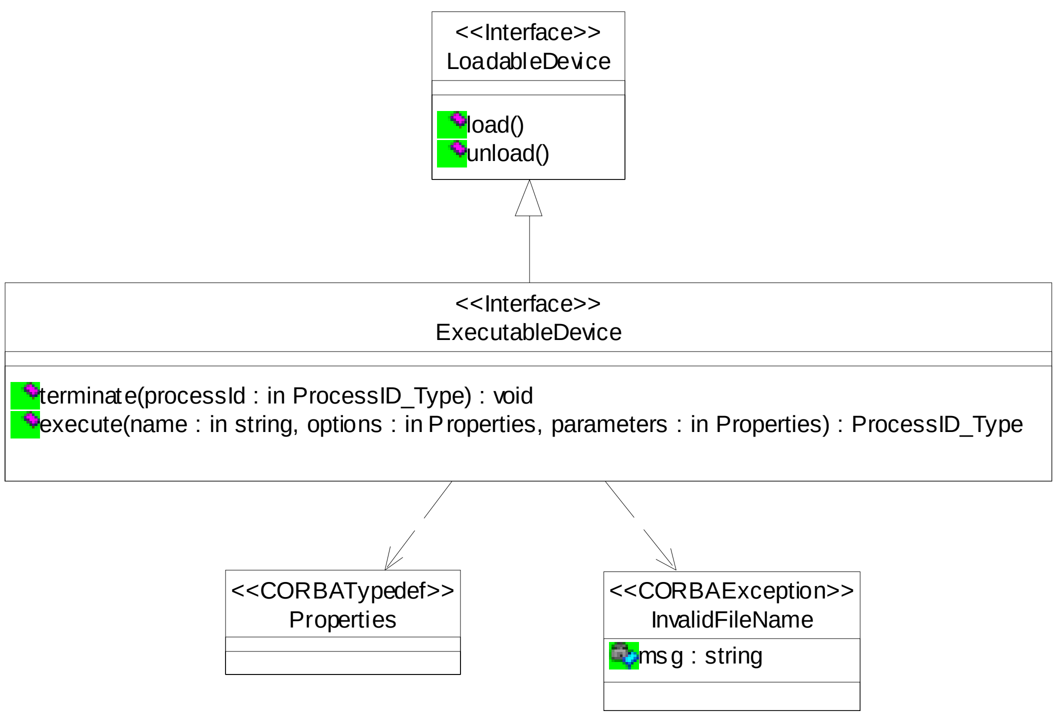

Figure 3-27. *ExecutableDevice* Interface UML

3.1.3.2.6.3 Types.

3.1.3.2.6.3.1 InvalidProcess.

The InvalidProcess exception indicates that a process, as identified by the processID parameter, does not exist on this device.

```
exception InvalidProcess {};
```

3.1.3.2.6.3.2 InvalidFunction.

The InvalidFunction exception indicates that a function, as identified by the input name parameter, hasn't been loaded on this device.

```
exception InvalidFunction {};
```

3.1.3.2.6.3.3 ProcessID_Type.

This type defines a process number within the system. Process number is unique to the Processor operating system that created the process.

```
typedef unsigned long ProcessID_Type;
```

3.1.3.2.6.3.4 InvalidParameters.

The InvalidParameters exception indicates the input parameters are invalid on the execute operation. The InvalidParameters exception is raised when there are invalid execute parameters. Each parameter's ID and value must be a valid string type. The invalidParms is a list of invalid parameters specified in the execute operation.

```
exception InvalidParameters { Properties invalidParms;};
```

3.1.3.2.6.3.5 InvalidOptions.

The InvalidOptions exception indicates the input options are invalid on the execute operation. The invalidOpts is a list of invalid options specified in the execute operation.

exception InvalidOptions { Properties invalidOpts;};

3.1.3.2.6.3.6 STACK_SIZE_ID.

The STACK_SIZE_ID is the identifier for the *ExecutableDevice's* execute options parameter. The value for a stack size shall be an unsigned long.

```
Constant string STACK_SIZE_ID = “STACK_SIZE”;
```

3.1.3.2.6.3.7 PRIORITY_ID.

The PRIORITY_ID is the identifier for the *ExecutableDevice's* execute options parameters. The value for a priority shall be an unsigned long.

Constant string PRIORITY_ID = “PRIORITY”;

3.1.3.2.6.4 Attributes.

N/A.

3.1.3.2.6.5 Operations.

3.1.3.2.6.5.1 *execute*.

3.1.3.2.6.5.1.1 Brief Rationale.

The *execute* operation provides the mechanism for starting up and executing a software process/thread on a device.

3.1.3.2.6.5.1.2 Synopsis.

```
ProcessID_Type execute(in string name, in Properties options, in Properties
parameters)
  raises (InvalidState, InvalidFunction, InvalidParameters, InvalidOptions,
InvalidFileName);
```

3.1.3.2.6.5.1.3 Behavior.

The *execute* operation shall execute the function or file identified by the input name parameter using the input parameters and options parameters. Whether the input name parameter is a function or a file name is device-implementation-specific.

The *execute* operation shall convert the input parameters (id/value string pairs) parameter to the standard argv of the POSIX exec family of functions, where argv(0) is the function name. The *execute* operation shall map the input parameters parameter to argv starting at index 1 as follows, argv (1) maps to input parameters (0) id and argv (2) maps to input parameters (0) value and so forth. The *execute* operation passes argv through the operating system "execute" function.

The *execute* operation input options parameters are STACK_SIZE_ID and PRIORITY_ID. The *execute* operation shall use these options, when specified, to set the operating system's process/thread stack size and priority, for the executable image of the given input name parameter.

3.1.3.2.6.5.1.4 Returns.

The *execute* operation shall return a unique processID for the process that it created or a processID of minus 1 (-1) when a process is not created.

3.1.3.2.6.5.1.5 Exceptions/Errors.

The *execute* operation shall raise the InvalidState exception when the *Device's* adminState is not UNLOCKED or operationalState is DISABLED.

The *execute* operation shall raise the InvalidFunction exception when the function indicated by the input name parameter does not exist for the *Device*.

The *execute* operation shall raise the CF InvalidFileName exception when the file name indicated by the input name parameter does not exist for the *Device*.

The *execute* operation shall raise the InvalidParameters exception when the input parameters parameter item ID or value are not string types.

The *execute* operation shall raise the InvalidOptions exception when the input options parameter does not comply with sections 3.1.3.2.6.3.5 STACK_SIZE_ID and 3.1.3.2.6.3.6 PRIORITY_ID."

3.1.3.2.6.5.2 *terminate*.

3.1.3.2.6.5.2.1 Brief Rationale.

The *terminate* operation provides the mechanism for terminating the execution of a process/thread on a specific device that was started up with the execute operation.

3.1.3.2.6.5.2.2 Synopsis.

```
void terminate(in ProcessID_Type processId) raise ( InvalidProcess,
InvalidState );
```

3.1.3.2.6.5.2.3 Behavior.

The *terminate* operation shall terminate the execution of the process/thread designated by the processId input parameter on the *Device*.

3.1.3.2.6.5.2.4 Returns.

This operation does not return a value.

3.1.3.2.6.5.2.5 Exceptions/Errors.

The *terminate* operation shall raise the InvalidState exception when the *Device's* adminState is LOCKED or operationalState is DISABLED.

The *terminate* operation shall raise the InvalidProcess exception when the processId does not exist for the *Device*.

3.1.3.2.7 *AggregateDevice*.

3.1.3.2.7.1 Description.

This interface defines aggregate behavior for a *Device* that can be used for adding and removing *Devices* from the *Device*. This new interface can be provided via inheritance or as a "provides port" for any *Device* that is capable of an aggregate relationship. Aggregated *Devices* use this interface to add or remove themselves from composite *Devices* when being created or torn-down.

3.1.3.2.7.2 UML.

The *AggregateDevice* Interface UML is depicted in Figure 3-28.

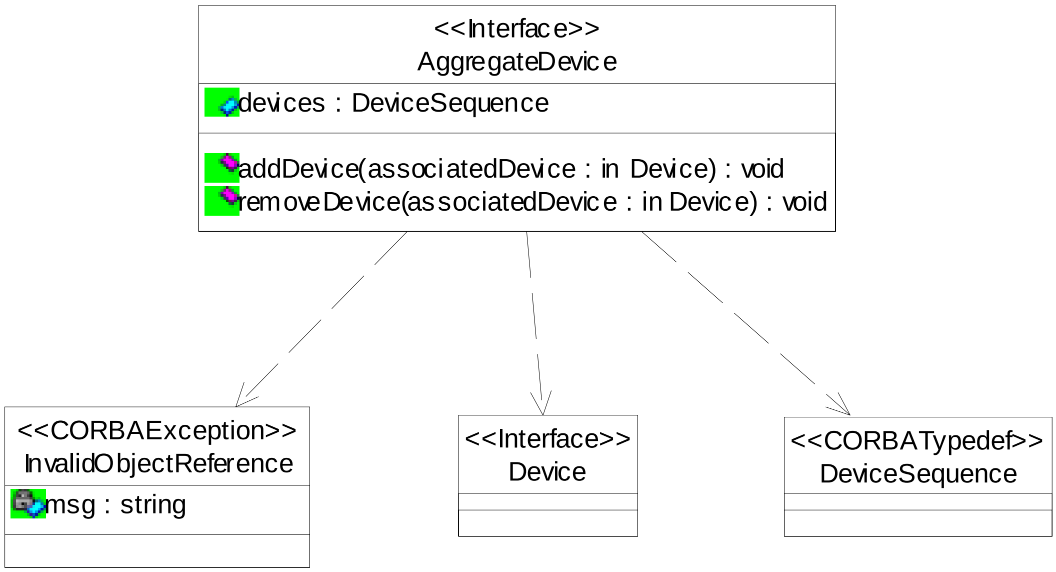

Figure 3-28. *AggregateDevice* Interface UML

3.1.3.2.7.3 Types.

N/A.

3.1.3.2.7.4 Attributes.

3.1.3.2.7.4.1 devices.

The readonly devices attribute shall contain a list of devices that have been added to this *Device* or a sequence length of zero if the *Device* has no aggregation relationships with other *Devices*.

```
readonly attribute DeviceSequence devices;
```

3.1.3.2.7.5 Operations.

3.1.3.2.7.5.1 *addDevice*.

3.1.3.2.7.5.1.1 Brief Rationale.

The *addDevice* operation provides the mechanism to associate a *Device* with another *Device*. When a *Device* changes state or it is being torn down, this affects its associated *Devices*.

3.1.3.2.7.5.1.2 Synopsis.

```
void addDevice(in Device associatedDevice) raises ( InvalidObjectReference );
```

3.1.3.2.7.5.1.3 Behavior.

The *addDevice* operation shall add the input associatedDevice parameter to the *AggregateDevice's* devices attribute when the associatedDevice does not exist in the devices attribute. The associatedDevice is ignored when duplicated.

The *addDevice* operation shall write a FAILURE_ALARM log record, upon unsuccessful adding of an associatedDevice to the *AggregateDevice's* devices attribute.

3.1.3.2.7.5.1.4 *Returns*.

This operation does not return any *value*.

3.1.3.2.7.5.1.5 Exceptions/Errors.

The *addDevice* operation shall raise the CF InvalidObjectReference when the input associatedDevice is a nil CORBA object reference.

3.1.3.2.7.5.2 *removeDevice*.

3.1.3.2.7.5.2.1 Brief Rationale.

The *removeDevice* operation provides the mechanism to disassociate a *Device* with another *Device*.

3.1.3.2.7.5.2.2 Synopsis.

```
void removeDevice(in Device associatedDevice) raises ( InvalidObjectReference
);
```

3.1.3.2.7.5.2.3 Behavior.

The *removeDevice* operation shall remove the input associatedDevice parameter from the *AggregateDevice's* devices attribute.

The *removeDevice* operation shall write a FAILURE_ALARM log record, upon unsuccessful removal of the associatedDevice from the *AggregateDevice's* devices attribute.

3.1.3.2.7.5.2.4 Returns.

This operation does not return any value.

3.1.3.2.7.5.2.5 Exceptions/Errors.

The *removeDevice* operation shall raise the CF InvalidObjectReference when the input associatedDevice is a nil CORBA object reference or does not exist in the *AggregateDevice's* devices attribute.

3.1.3.2.8 *DeviceManager*.

3.1.3.2.8.1 Description.

The *DeviceManager* interface is used to manage a set of logical *Devices* and services. The interface for a *DeviceManager* is based upon its attributes, which are:

1. Device Configuration Profile - a mapping of physical device locations to meaningful labels (e.g., audio1, serial1, etc.), along with the *Devices* and services to be deployed.
2. File System - the FileSystem associated with this *DeviceManager*.
3. Device Manager Identifier - the instance-unique identifier for this *DeviceManager*.
4. Device Manager Label - the meaningful name given to this *DeviceManager*.
5. Registered Devices - a list of *Devices* that have registered with this *DeviceManager*.
6. Registered Services - a list of Services that have registered with this *DeviceManager*

3.1.3.2.8.2 UML.

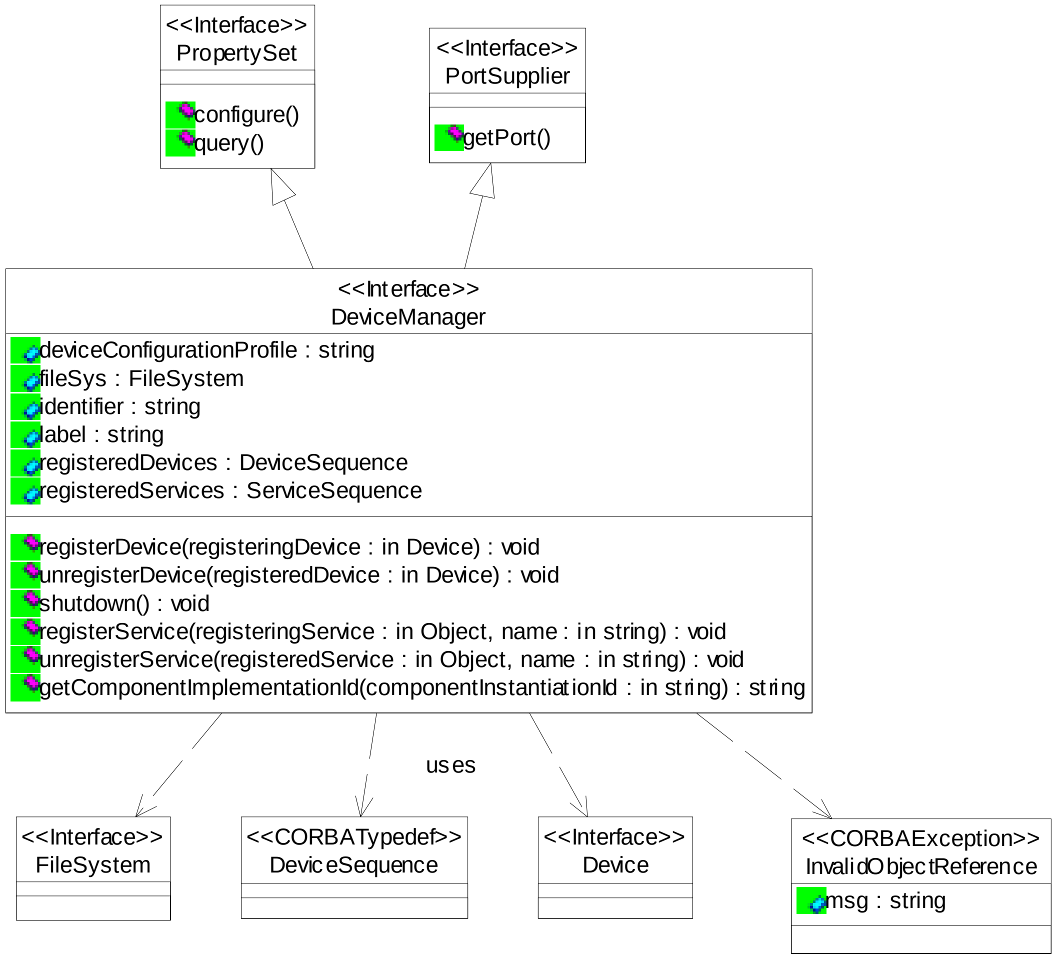

Figure 3-29. *DeviceManager* UML

3.1.3.2.8.3 Types.

This section describes the types defined in the interface *DeviceManager*.

3.1.3.2.8.3.1 ServiceType.

This structure provides the object reference and name of services that have registered with the *DeviceManager*.

```
struct ServiceType{
     Object serviceObject;
     string serviceName;
   };
```

3.1.3.2.8.3.2 ServiceSequenceType.

This type provides an unbounded sequence of ServiceType structures for services that have registered with the *DeviceManager*.

```
typedef sequence <ServiceType> ServiceSequence;
```

3.1.3.2.8.4 Attributes.

3.1.3.2.8.4.1 identifier.

The readonly identifier attribute shall contain the instance-unique identifier for a *DeviceManager*.

```
readonly attribute string identifier;
```

3.1.3.2.8.4.2 label.

The readonly label attribute shall contain the *DeviceManager's* label. The label attribute is the meaningful name given to a *DeviceManager*.

```
readonly attribute string label;
```

3.1.3.2.8.4.3 filesys.

The readonly fileSys attribute shall contain the *FileSystem* associated with this *DeviceManager* or a nil CORBA object reference if no *FileSystem* is associated with this *DeviceManager*.

```
readonly attribute FileSystem fileSys;
```

3.1.3.2.8.4.4 deviceConfigurationProfile.

The readonly deviceConfigurationProfile attribute contains the *DeviceManager's* profile.

The readonly deviceConfigurationProfile attribute shall contain either a profile element with a file reference to the *DeviceManager's* device configuration (DCD) profile or the XML for the *DeviceManager's* device configuration (DCD) profile. Files referenced within the profile are obtained from a *FileSystem*.

```
readonly attribute string deviceConfigurationProfile;
```

3.1.3.2.8.4.5 registeredDevices.

The readonly registeredDevices attribute shall contain a list of *Devices* that have registered with this *DeviceManager* or a sequence length of zero if no *Devices* have registered with the *DeviceManager*.

```
readonly attribute DeviceSequence registeredDevices;
```

3.1.3.2.8.4.6 registeredServices.

The readonly registeredServices attribute shall contain a list of Services that have registered with this *DeviceManager* or a sequence length of zero if no Services have registered with the *DeviceManager*.

```
readonly attribute ServiceSequence registeredServices;
```

3.1.3.2.8.5 General Behavior.

The *DeviceManager* upon start up shall register itself with a *DomainManager*. This requirement allows the system to be developed where at a minimum only the *DomainManager's* component reference needs to be known. A *DeviceManager* shall use the *DeviceManager's* deviceConfigurationProfile attribute for determining:

1. How to obtain the *DomainManager* component reference, whether Naming Service is being used or a *DomainManager* stringified IOR is being used (see also 3.1.3.2.2.5.1.3),
2. Services to be deployed for this *DeviceManager* (for example, *log*(s)),
3. *Devices* to be created for this *DeviceManager* (when the DCD *deployondevice* element is not specified then the DCD *componentinstantiation* element is deployed on the same hardware device as the *DeviceManager*),
4. *Devices* to be deployed on (executing on) another *Device*,
5. *Devices* to be aggregated to another *Device*,
6. Mount point names for *FileSystems*,
7. The DCD's id attribute for the *DeviceManager's* identifier attribute value, and
8. The DCD's name attribute for the *DeviceManager's* label attribute value.

The *DeviceManager* shall create *FileSystem* components implementing the *FileSystem* interface for each OS file system. If multiple *FileSystems* are to be created, the *DeviceManager* shall mount created *FileSystems* to a *FileManager* component (widened to a *FileSystem* through the FileSys attribute). Each mounted *FileSystem* name must be unique within the *DeviceManager*.

The *DeviceManager* shall supply *execute* operation parameters (IDs and format values) for a *Device* consisting of:

A. *DeviceManager* IOR – The ID is "DEVICE_MGR_IOR" and the value is a string that is the *DeviceManager* stringified IOR.

B. Profile Name – The ID is "PROFILE_NAME" and the value is a CORBA string that is the full mounted file system file path *name*.

C. Device Identifier – The ID is "DEVICE_ID" and the value is a string that corresponds to the DCD componentinstantiation id attribute.

D. Device Label – The ID is "DEVICE_LABEL" and the value is a string that corresponds to the DCD componentinstantiation usage element. This parameter is only used when the DCD componentinstantiation usage element is specified.

E. Composite Device IOR - The ID is "Composite_DEVICE_IOR" and the value is a string that is an *AggregateDevice* stringified IOR. This parameter is only used when the DCD componentinstantiation element is a composite part of another componentinstantiation element.

F. The execute ("execparam") properties as specified in the DCD for a componentinstantiation element. The *DeviceManager* shall pass the componentinstantiation element "execparam" properties that have values as parameters. The *DeviceManager* shall pass “execparam” parameters’ IDs and values as string values.

The *DeviceManager* shall use the componentinstantiation element’s SPD implementation code’s *stacksize* and *priority* elements, when specified, for the *execute* options parameters.

The *DeviceManager* shall initialize and configure logical *Devices* that are started by the *DeviceManager* after they have registered with the *DeviceManager*. The *DeviceManager* shall configure a DCD’s componentinstantiation element provided the componentinstantiation element has “configure” readwrite or writeonly properties with values. Figure 3-30 depicts a *DeviceManager* startup scenario. If a Service is deployed by the *DeviceManager*, the *DeviceManager* shall supply execute operation parameters (IDs and format values) consisting of:

a. *DeviceManager* IOR – The ID is “DEVICE_MGR_IOR” and the value is a string that is the *DeviceManager* stringified IOR.

b. Service Name – The ID is “SERVICE_NAME” and the value is a string that corresponds to the DCD componentinstantiation usagename element.

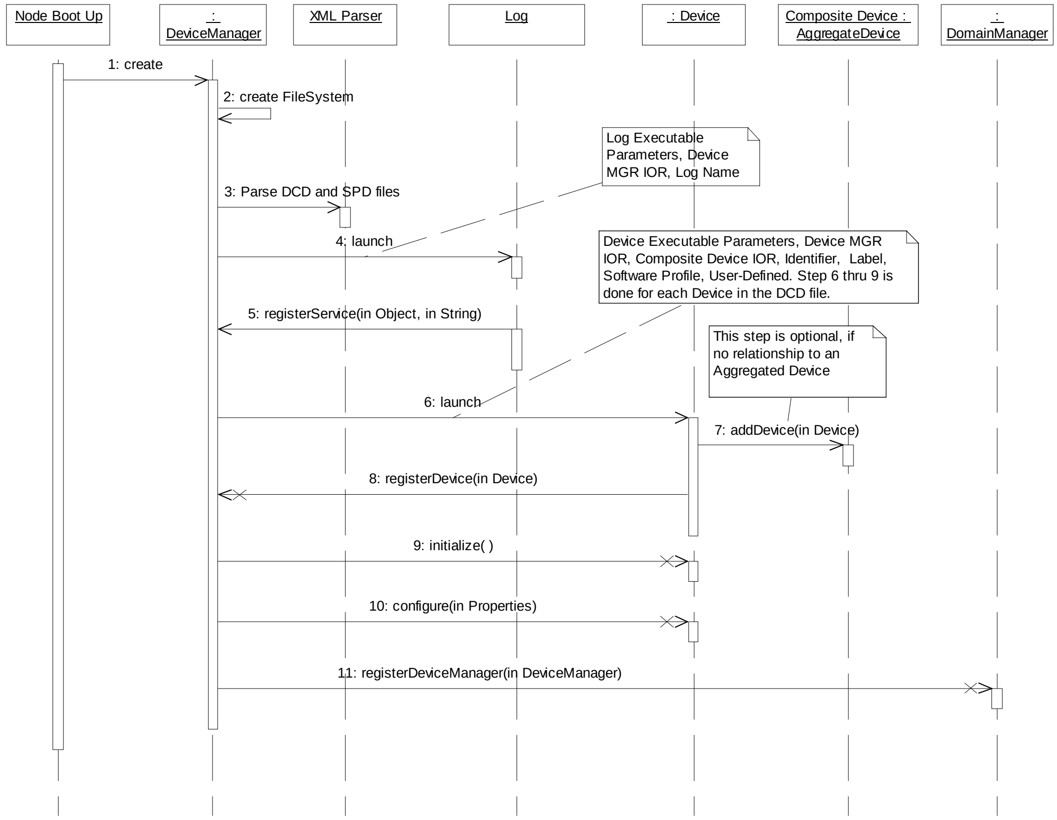

Figure 3-30. *DeviceManager* Startup Scenario

3.1.3.2.8.6 Operations.

3.1.3.2.8.6.1 registerDevice.

3.1.3.2.8.6.1.1 Brief Rationale.

The *registerDevice* operation provides the mechanism to register a *Device* with a *DeviceManager*.

3.1.3.2.8.6.1.2 Synopsis.

```
void registerDevice(in Device registeringDevice) raises (
InvalidObjectReference );
```

3.1.3.2.8.6.1.3 Behavior.

The *registerDevice* operation shall add the input registeringDevice to the *DeviceManager's* registeredDevices attribute when the input registeringDevice does not already exist in the registeredDevices attribute. The registeringDevice is ignored when duplicated.

The *registerDevice* operation shall register the registeringDevice with the *DomainManager* when the *DeviceManager* has already registered to the *DomainManager* and the registeringDevice has been successfully added to the *DeviceManager's* registeredDevices attribute.

The *registerDevice* operation shall write a FAILURE_ALARM log record to a *DomainManager's* Log, upon unsuccessful registration of a *Device* to the *DeviceManager's* registeredDevices.

3.1.3.2.8.6.1.4 Returns.

This operation does not return any value.

3.1.3.2.8.6.1.5 Exceptions/Errors.

The *registerDevice* operation shall raise the CF InvalidObjectReference when the input registeredDevice is a nil CORBA object reference.

3.1.3.2.8.6.2 unregisterDevice.

3.1.3.2.8.6.2.1 Brief Rationale.

The *unregisterDevice* operation unregisters a *Device* from a *DeviceManager*.

3.1.3.2.8.6.2.2 Synopsis.

```
void unregisiterDevice(in Device registeredDevice) raises (
InvalidObjectReference );
```

3.1.3.2.8.6.2.3 Behavior.

The *unregisterDevice* operation shall remove the input registeredDevice from the *DeviceManager's* registeredDevices attribute. The *unregissiterDevice* operation shall unregister the input registeredDevice from the *DomainManager* when the input registeredDevice is registered with the *DeviceManager* and the *DeviceManager* is not shutting down.

The *unregisterDevice* operation shall write a FAILURE_ALARM log record, when it cannot successfully remove a registeredDevice from the *DeviceManager*'s registeredDevices.

3.1.3.2.8.6.2.4 Returns.

This operation does not return any value.

3.1.3.2.8.6.2.5 Exceptions/Errors.

The *unregisterDevice* operation shall raise the CF InvalidObjectReference when the input registeredDevice is a nil CORBA object reference or does not exist in the *DeviceManager's* registeredDevices attribute.

3.1.3.2.8.6.3 registerService.

3.1.3.2.8.6.3.1 Brief Rationale.

The *registerService* operation provides the mechanism to register a Service with a *DeviceManager*.

3.1.3.2.8.6.3.2 Synopsis.

```
void registerService(in Object registeringService, in string name) raises (
InvalidObjectReference );
```

3.1.3.2.8.6.3.3 Behavior.

The *registerService* operation shall add the input registeringService to the *DeviceManager's* registeredServices attribute when the input registeringService does not already exist in the registeredServices attribute. The registeringService is ignored when duplicated.

The *registerService* operation shall register the registeringService with the *DomainManager* when the *DeviceManager* has already registered to the *DomainManager* and the registeringService has been successfully added to the *DeviceManager's* registeredServices *attribute*.

The *registerService* operation shall write a FAILURE_ALARM log record, upon unsuccessful registration of a Service to the *DeviceManager*'s *registeredServices*.

3.1.3.2.8.6.3.4 Returns.

This operation does not return any value.

3.1.3.2.8.6.3.5 Exceptions/Errors.

The *registerService* operation shall raise the CF InvalidObjectReference exception when the input registeredService is a nil CORBA object reference.

3.1.3.2.8.6.4 unregisterService.

3.1.3.2.8.6.4.1 Brief Rationale.

The *unregisterService* operation unregisters a Service from a *DeviceManager*.

3.1.3.2.8.6.4.2 Synopsis.

```
void unregisterService(in Object registeredS ervice) raises (
InvalidObjectReference );
```

3.1.3.2.8.6.4.3 Behavior.

The *unregisterService* operation shall remove the input registeredService from the *DeviceManager's* registeredServices attribute. The *unregisterService* operation shall unregister the input registeredService from the *DomainManager* when the input *registeredService* is registered with the *DeviceManager* and the *DeviceManager* is not in the shutting down state.

The *unregisterService* operation shall write a FAILURE_ALARM log record, when it cannot successfully remove a registeredService from the *DeviceManager*'s registeredServices.

3.1.3.2.8.6.4.4 Returns.

This operation does not return any value.

3.1.3.2.8.6.4.5 Exceptions/Errors.

The *unregisterService* operation shall raise the CF InvalidObjectReference when the input registeredService is a nil CORBA object reference or does not exist in the *DeviceManager*'s registeredServices *attribute*.

3.1.3.2.8.6.5 *shutdown*.

3.1.3.2.8.6.5.1 Brief Rationale.

The *shutdown* operation provides the mechanism to terminate a *DeviceManager*.

3.1.3.2.8.6.5.2 Synopsis.

```
void shutdown();
```

3.1.3.2.8.6.5.3 Behavior.

The *shutdown* operation shall unregister the *DeviceManager* from the *DomainManager*.

The *shutdown* operation shall perform releaseObject on all of the *DeviceManager's* registered *Devices* (*DeviceManager's* registeredDevices attribute).

The *shutdown* operation shall cause the *DeviceManager* to be unavailable (i.e. released from the CORBA environment and its process terminated on the OS), when all of the *DeviceManager's* registered *Devices* are unregistered from the *DeviceManager*.

3.1.3.2.8.6.5.4 Returns.

This operation does not return any value.

3.1.3.2.8.6.5.5 Exceptions/Errors.

This operation does not raise any exceptions.

3.1.3.2.8.6.6 *getComponentImplementationId*.

3.1.3.2.8.6.6.1 Brief Rational.

The *getComponentImplementationId* operation returns the SPD implementation ID that the *DeviceManager* interface used to create a component.

3.1.3.2.8.6.6.2 Synopsis.

```
string getComponentImplementationId (in string componentInstantiationId);
```

3.1.3.2.8.6.6.3 Behavior.

The *getComponentImplementationId* operation will return the SPD implementation ID attribute that matches the SPD implementation that was used to create the component as identified by the input componentInstantiation ID *parameter*.

3.1.3.2.8.6.6.4 Returns.

The *getComponentImplementationId* operation shall return the SPD implementation ID attribute that matches the SPD implementation which was used to create the component as identified by the input componentInstantiation ID parameter. The getComponentImplementationId operation shall return an empty string when the componentInstantiation is invalid.

3.1.3.2.8.6.6.5 Exceptions/Errors.

This operation does not raise any exceptions.

3.1.3.3 Framework Services Interfaces.

Framework Services Interfaces shall be implemented using the CF IDL presented in Appendix C.

3.1.3.3.1 *File*.

3.1.3.3.1.1 Description.

The *File* interface provides the ability to read and write files residing within a CF-compliant, distributed *FileSystem*. A file can be thought of conceptually as a sequence of octets with a current filepointer describing where the next read or write will occur. This filepointer points to the beginning of the file upon construction of the file object. The *File* interface is modeled after the POSIX/C file interface. (Reference *File* Interface UML in Figure 3-31.)

3.1.3.3.1.2 UML.

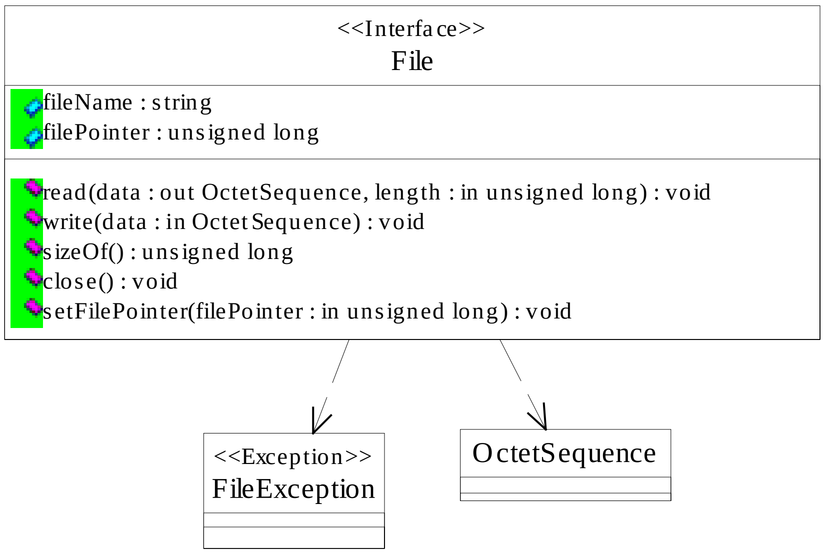

Figure 3-31. *File* Interface UML

3.1.3.3.1.3 Types.

3.1.3.3.1.3.1 IOException.

The IOException exception indicates an error occurred during a *read* or *write* operation to a *File*. The message is component-dependent, providing additional information describing the reason why the error occurred.

```
exception IOException { string msg; unsigned short errorCode};
```

3.1.3.3.1.3.2 InvalidFilePointer.

The InvalidFilePointer exception indicates the file pointer is out of range based upon the current file size.

```
exception InvalidFilePointer {};
```

3.1.3.3.1.4 Attributes.

3.1.3.3.1.4.1 fileName.

The readonly fileName attribute shall contain the file name given to the *FileSystem open*/*create* operation. The syntax for a filename is based upon the UNIX operating system. That is, a sequence of directory names separated by forward slashes (/) followed by the base filename. The fileName attribute will contain the filename given to the *FileSystem open* operation.

```
readonly attribute string fileName;
```

3.1.3.3.1.4.2 filePointer.

The readonly filePointer attribute shall contain the file position where the next read or write will occur.

```
readonly attribute unsigned long filePointer;
```

3.1.3.3.1.5 Operations.

3.1.3.3.1.5.1 *read*.

3.1.3.3.1.5.1.1 Brief Rationale.

Applications require the *read* operation in order to retrieve data from remote files.

3.1.3.3.1.5.1.2 Synopsis.

```
void read(out OctetSequence data, in unsigned long length) raises (
IOException);
```

3.1.3.3.1.5.1.3 Behavior.

The *read* operation shall read octets from the file referenced up to the number specified by the length parameter and change the value of the filePointer attribute forward by the number of octets actually read. The *read* operation shall only read less than the maximum number of octets specified in the length parameter when an end of file is encountered.

3.1.3.3.1.5.1.4 Returns.

The *read* operation shall return via the out Message parameter a CF OctetSequence that equals the number of octets actually read from the *File*. If the filePointer attribute value reflects the end of the *File*, the *read* operation shall return a 0-length CF OctetSequence.

3.1.3.3.1.5.1.5 Exceptions/Errors.

The *read* operation shall raise the IOException when a *read* error occurs.

3.1.3.3.1.5.2 *write*.

3.1.3.3.1.5.2.1 Brief Rationale.

Applications require the *write* operation in order to write data to remote files.

3.1.3.3.1.5.2.2 Synopsis.

```
void write(in OctetSequence data) raises ( IOException);
```

3.1.3.3.1.5.2.3 Behavior.

The *write* operation shall write data to the file referenced. If the write is successful, the *write* operation shall increment the filePointer attribute to reflect the number of octets written. If the *write* is unsuccessful, the filePointer attribute value shall maintain or be restored to its value prior to the *write* operation call.

3.1.3.3.1.5.2.4 Returns.

This operation does not return any value.

3.1.3.3.1.5.2.5 Exceptions/Errors.

The *write* operation shall raise the IOException when a write error occurs.

3.1.3.3.1.5.3 sizeOf.

3.1.3.3.1.5.3.1 Brief Rationale.

An application may need to know the size of a file in order to determine memory allocation requirements.

3.1.3.3.1.5.3.2 Synopsis.

```
unsigned long sizeOf() raises ( FileException );
```

3.1.3.3.1.5.3.3 Behavior.

There is no significant behavior beyond the behavior described by the following section.

3.1.3.3.1.5.3.4 Returns.

The *sizeOf* operation shall return the number of octets stored in the file.

3.1.3.3.1.5.3.5 Exceptions/Errors.

The *sizeOf* operation shall raise the CF FileException when a file-related error occurs (e.g., file does not exist anymore).

3.1.3.3.1.5.4 *close*.

3.1.3.3.1.5.4.1 Brief Rationale.

The *close* operation is needed in order to release file resources once they are no longer needed.

3.1.3.3.1.5.4.2 Synopsis.

```
void close() raises ( FileException );
```

3.1.3.3.1.5.4.3 Behavior.

The *close* operation shall release any OE file resources associated with the component. The *close* operation shall make the file unavailable to the component.

3.1.3.3.1.5.4.4 Returns.

This operation does *not* return any value.

3.1.3.3.1.5.4.5 Exceptions/Errors.

The *close* operation shall raise the CF FileException when it can not successfully close the file.

3.1.3.3.1.5.5 *setFilePointer*.

3.1.3.3.1.5.5.1 Brief Rationale.

The *setFilePointer* operation positions the file pointer where the next read or write will occur.

3.1.3.3.1.5.5.2 Synopsis.

```
void setFilePointer(in unsigned long filePointer) raises (
InvalidFilePointer, FileException );
```

3.1.3.3.1.5.5.3 Behavior.

The *setFilePointer* operation shall set the filePointer attribute value to the input filePointer.

3.1.3.3.1.5.5.4 Returns.

This operation does not return anyvalue.

3.1.3.3.1.5.5.5 Exceptions/Errors.

The *setFilePointer* operation shall raise the CF FileException when the *File* can not be successfully accessed to set thefilePointer attribute. The *setFilePointer* operation shall raise the InvalidFilePointer exception when the value of the filePointer parameter exceeds the file size.

3.1.3.3.2 *FileSystem*.

3.1.3.3.2.1 Description.

The *FileSystem* interface defines CORBA operations that enable remote access to a physical file system. (Reference *FileSystem* interface UML in Figure 3-32.)

3.1.3.3.2.2 UML.

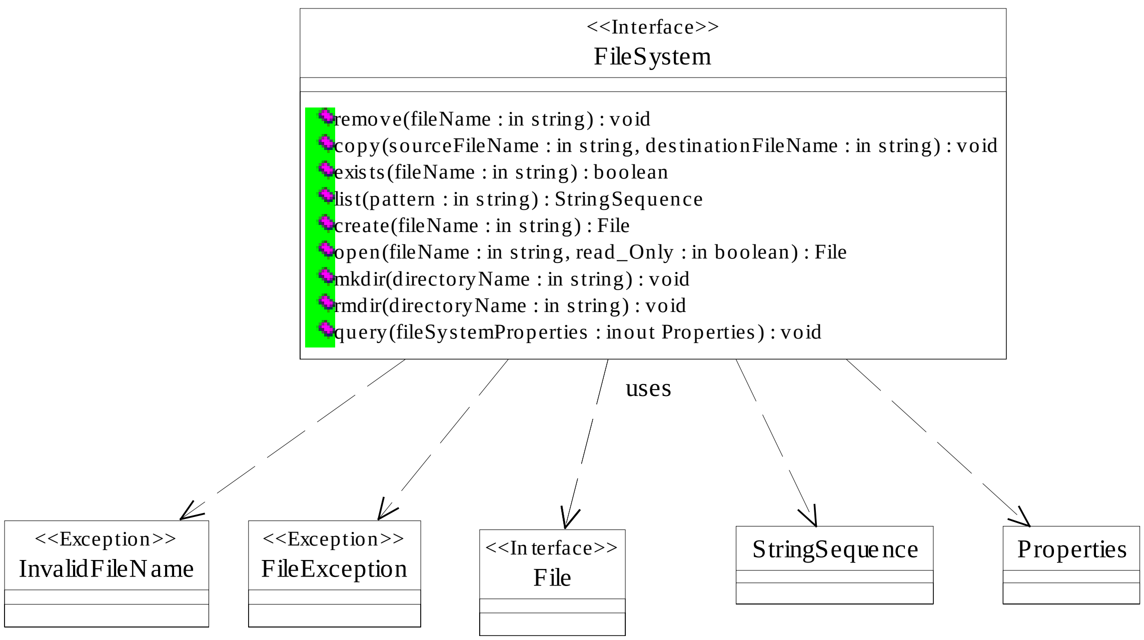

Figure 3-32. *FileSystem* Interface UML

3.1.3.3.2.3 Types.

3.1.3.3.2.3.1 UnknownFileSystemProperties.

The UnknownFileSystemProperties exception indicates a set of properties unknown by the component.

```
exception UnknownFileSystemProperties {properties invalidProperties; };
```

3.1.3.3.2.3.2 fileSystemProperties Query Constants.

Constants are defined to be used for the *query* operation (see section 3.1.3.3.2.5.9).

```
const string SIZE = “SIZE”;
const string AVAILABLE_SPACE = “AVAILABLE_SPACE”;
```

3.1.3.3.2.4 Attributes.

N/A.

3.1.3.3.2.5 Operations.

3.1.3.3.2.5.1 *remove*.

3.1.3.3.2.5.1.1 Brief Rationale.

The *remove* operation provides the ability to remove a file from a file system.

3.1.3.3.2.5.1.2 Synopsis.

```
void remove(in string fileName) raises( FileException, InvalidFileName );
```

3.1.3.3.2.5.1.3 Behavior.

The *remove* operation shall remove the file with the given filename.

3.1.3.3.2.5.1.4 Returns.

This operation does not return any value.

3.1.3.3.2.5.1.5 Exceptions/Errors.

The *remove* operation shall raise the CF InvalidFilename exception when the filename is not a valid filename or not an absolute pathname.

The *remove* operation shall raise the CF FileException when a file-related error occurs.

3.1.3.3.2.5.2 *copy*.

3.1.3.3.2.5.2.1 Brief Rationale.

The *copy* operation provides the ability to copy a file to another file.

3.1.3.3.2.5.2.2 Synopsis.

```
void copy(in string sourceFileName, in string destinationFileName) raises(
InvalidFileName, FileException );
```

3.1.3.3.2.5.2.3 Behavior.

The *copy* operation shall copy the source file with the specified sourceFileName to the destination file with the specified destinationFileName.

3.1.3.3.2.5.2.4 Returns.

This operation does not return any value.

3.1.3.3.2.5.2.5 Exceptions/Errors.

The *copy* operation shall raise the CF FileException when a file-related error occurs.

The *copy* operation shall raise the CF InvalidFilename exception when the filename is not a valid file name or not an absolute pathname.

3.1.3.3.2.5.3 *exists*.

3.1.3.3.2.5.3.1 Brief Rationale.

The *exists* operation provides the ability to verify the existence of a file within a *FileSystem*.

3.1.3.3.2.5.3.2 Synopsis.

```
boolean exists(in string fileName) raises( InvalidFileName );
```

3.1.3.3.2.5.3.3 Behavior.

The *exists* operation shall check to see if a file exists based on the fileName parameter.

3.1.3.3.2.5.3.4 Returns.

The *exists* operation shall return True if the file exists, or False if it does not.

3.1.3.3.2.5.3.5 Exceptions/Errors.

The *exists* operation shall raise the CF InvalidFilename exception when fileName is not a valid file name or not an absolute pathname.

3.1.3.3.2.5.4 *list*.

3.1.3.3.2.5.4.1 Brief Rationale.

The *list* operation provides the ability to obtain a list of files in the *FileSystem* according to a given search pattern.

3.1.3.3.2.5.4.2 Synopsis.

```
StringSequence list(in string pattern)raises (InvalidFileName);
```

3.1.3.3.2.5.4.3 Behavior.

The *list* operation shall return a list of filenames based upon the search pattern given. The *list* operation shall support the following wildcard characters:

(1) * used to match any sequence of characters (including null).

(2) ? used to match any single character.

These wildcards may only be applied to the base filename in the search pattern given. For example, the following are valid search patterns:

/tmp/files/*.* Returns all files and directories within the /tmp/files directory. Directory names indicated with a “/” at the end of the name.
/tmp/files/foo* Returns all files beginning with the letters “foo” in the /tmp/files directory.
/tmp/files/f?? Returns all 3 letter files beginning with the letter f in the /tmp/files directory.
/*/files/foo* Returns all files in subdirectories of the name “files” and starting with the letters “foo”.
/*/fi?es/f??.* Returns all files in subdirectories of the name “fi”, some character and “es” and starting with the letter “f” followed by any 2 characters.

3.1.3.3.2.5.4.4 Returns.

The *list* operation shall return a CF StringSequence of filenames matching the wildcard specification.

3.1.3.3.2.5.4.5 Exceptions/Errors.

The *list* operation shall raise the CF InvalidFileName exception when the input pattern does not start with a slash "/" or cannot be interpreted due to unexpected characters.

3.1.3.3.2.5.5 *create*.

3.1.3.3.2.5.5.1 Brief Rationale.

The *create* operation provides the ability to create a new file on the *FileSystem*.

3.1.3.3.2.5.5.2 Synopsis.

```
File create(in string fileName) raises( InvalidFileName, FileException );
```

3.1.3.3.2.5.5.3 Behavior.

The *create* operation shall create a new *File* based upon the provided fileName attribute.

3.1.3.3.2.5.5.4 Returns.

The *create* operation shall return a *File* component reference to the opened file. The *create* operation shall return a null file component reference if an error occurs.

3.1.3.3.2.5.5.5 Exceptions/Errors.

The *create* operation shall raise the CF FileException if the file already exists or another file error occurred.

The *create* operation shall raise the CF InvalidFilename exception when a fileName is not a valid file name or not an absolute pathname.

3.1.3.3.2.5.6 *open*.

3.1.3.3.2.5.6.1 Brief Rationale.

The *open* operation provides the ability to open a file for read or write.

3.1.3.3.2.5.6.2 Synopsis.

```
File open(in string fileName, in boolean read_Only) raises( InvalidFileName,
FileException );
```

3.1.3.3.2.5.6.3 Behavior.

The *open* operation shall open a file based upon the input fileName. The read_Only parameter indicates if the file should be opened for read access only. The *open* operation shall open the file for write access when the read_Only parameter is false.

3.1.3.3.2.5.6.4 Returns.

The *open* operation shall return a *File* component parameter on successful completion. The *open* operation shall return a null file component reference if the *open* operation is unsuccessful. If the file is opened with the read_Only flag set to true, then writes to the file will be considered an error.

3.1.3.3.2.5.6.5 Exceptions/Errors.

The *open* operation shall raise the CF FileException if the file does not exist or another file error occurred.

The *open* operation shall raise the CF InvalidFilename exception when the filename is not a valid file name or not an absolute pathname.

3.1.3.3.2.5.7 *mkdir*.

3.1.3.3.2.5.7.1 Brief Rationale.

The *mkdir* operation provides the ability to create a directory on the file system.

3.1.3.3.2.5.7.2 Synopsis.

```
void mkdir(in string directoryName) raises( InvalidFileName, FileException );
```

3.1.3.3.2.5.7.3 Behavior.

The *mkdir* operation shall create a *FileSystem* directory based on the directoryName given. The *mkdir* operation shall create all parent directories required to create the directoryName path given.

3.1.3.3.2.5.7.4 Returns.

This operation does not return any value.

3.1.3.3.2.5.7.5 Exceptions/Errors.

The *mkdir* operation shall raise the CF FileException if a file-related error occurred during the operation.

The *mkdir* operation shall raise the CF InvalidFilename exception when the directoryName is not a valid directory name.

3.1.3.3.2.5.8 *rmdir*.

3.1.3.3.2.5.8.1 Brief Rationale.

The *rmdir* operation provides the ability to remove a directory from the file system.

3.1.3.3.2.5.8.2 Synopsis.

```
void rmdir(in string directoryName) raises( InvalidFileName, FileException );
```

3.1.3.3.2.5.8.3 Behavior.

The *rmdir* operation shall remove a *FileSystem* directory, based on the directoryName given, only if the directory is empty (no files exist in directory).

3.1.3.3.2.5.8.4 Returns.

This operation does not return any value.

3.1.3.3.2.5.8.5 Exceptions/Errors.

The *rmdir* operation shall raise the CF FileException when the directory does not exist, if the directory is not empty, or another file-related error occurred.

The *rmdir* operation shall raise the CF InvalidFilename exception when the directoryName is not a valid directory name.

3.1.3.3.2.5.9 *query*.

3.1.3.3.2.5.9.1 Brief Rationale.

The *query* operation provides the ability to retrieve information about a file system.

3.1.3.3.2.5.9.2 Synopsis.

```
void query(inout Properties fileSystemProperties) raises(
UnknownFileSystemProperties );
```

3.1.3.3.2.5.9.3 Behavior.

The *query* operation shall return file system information to the calling client based upon the given fileSystemProperties' ID.

As a minimum, the *FileSystem query* operation shall support the following fileSystemProperties:

1. SIZE – an ID value of “SIZE causes query to return an unsigned long long containing the file system size (in octets).
2. AVAILABLE SPACE – an ID value of “AVAILABLE SPACE” causes the query operation to return an unsigned long long containing the available space on the file system (in octets),

See section 3.1.3.3.2.3.2 for the constants for the fileSystemProperties.

3.1.3.3.2.5.9.4 Returns.

This operation does not return any value.

3.1.3.3.2.5.9.5 Exceptions/Errors.

The *query* operation shall raise the UnknownFileSystemProperties exception when the given file system property is not recognized.

3.1.3.3.3 *FileManager*.

3.1.3.3.3.1 Description.

Multiple, distributed *FileSystems* may be accessed through a *FileManager*. The *FileManager* interface appears to be a single *FileSystem* although the actual file storage may span multiple physical file systems. (Reference the *FileManager* interface UML in Figure 3-33.)

This is called a federated file system. A federated file system is created using the *mount* and *unmount* operations. Typically, the *DomainManager* or system initialization software will invoke these operations.

The *FileManager* inherits the IDL interface of a *FileSystem*. Based upon the pathname of a directory or file and the set of mounted *FileSystems*, the *FileManager* will delegate the *FileSystem* operations to the appropriate *FileSystem*. For example, if a *FileSystem* is mounted at `/ppc2`, an *open* operation for a file called `/ppc2/profile.xml` would be delegated to the mounted *FileSystem*. The mounted *FileSystem* will be given the filename relative to it. In this example the *FileSystem's open* operation would receive `/profile.xml` as the fileName argument.

Another example of this concept can be shown using the *copy* operation. When a client invokes the *copy* operation, the *FileManager* will delegate operations to the appropriate *FileSystems* (based upon supplied pathnames) thereby allowing copy of files between *FileSystems*.

If a client does not need to mount and unmount *FileSystems*, it can treat the *FileManager* as a *FileSystem* by CORBA widening a *FileManager* reference to a *FileSystem* reference. One can always widen a *FileManager* to a *FileSystem* since the *FileManager* is derived from a *FileSystem*.

3.1.3.3.3.2 UML.

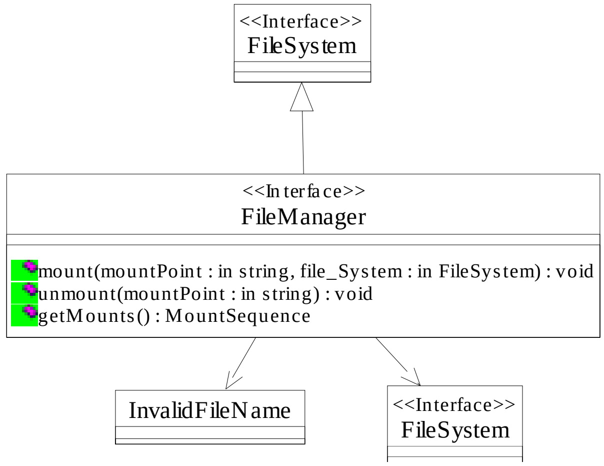

Figure 3-33. *FileManager* Interface UML

3.1.3.3.3.3 Types.

3.1.3.3.3.3.1 MountType.

The MountType structure shall identify the *FileSystems* mounted within the *FileManager*.

```
struct MountType {
  string mountPoint;
  FileSystem fs;
};
```

3.1.3.3.3.3.2 MountSequence.

The MountSequence is an unbounded sequence of Mount types.

```
typedef sequence <MountType> MountSequence;
```

3.1.3.3.3.3.3 NonExistentMount.

The NonExistentMount exception indicates a mount point does not exist within the *FileManager*.

```
exception NonExistentMount {};
```

3.1.3.3.3.3.4 MountPointAlreadyExists.

The MountPointAlreadyExists exception indicates the mount point is already in use in the *FileManager*.

```
exception MountPointAlreadyExists {};
```

3.1.3.3.3.3.5 InvalidFileSystem.

The InvalidFileSystem exception indicates the *FileSystem* is a null (nil) object reference.

```
exception InvalidFileSystem {};
```

3.1.3.3.3.4 Attributes.

N/A

3.1.3.3.3.5 Operations.

3.1.3.3.3.5.1 *mount*.

3.1.3.3.3.5.1.1 Brief Rationale.

The *FileManager* supports the notion of a federated file system. To create a federated file system, the *mount* operation associated a *FileSystem* with a mount point (a directory name).

3.1.3.3.3.5.1.2 Synopsis.

```
void mount(in string mountPoint, in FileSystem file_System) raises(
InvalidFileName, InvalidFileSystem, MountPointAlreadyExists );
```

3.1.3.3.3.5.1.3 Behavior.

The *mount* operation shall associate the specified *FileSystem* with the given mountPoint. A mountPoint name shall begin with a "/". A mountPoint name is a logical directory name for a *FileSystem*.

3.1.3.3.3.5.1.4 Returns.

This operation does not return any value.

3.1.3.3.3.5.1.5 Exceptions/Errors.

The *mount* operation shall raise the CF InvalidFileName exception when the input file name is invalid.

The *mount* operation shall raise the MountPointAlreadyExists exception when the mountPoint already exists in the file manager.

The *mount* operation shall raise the InvalidFileSystem exception when the input *FileSystem* is a null object reference.

3.1.3.3.3.5.2 *unmount*.

3.1.3.3.3.5.2.1 Brief Rationale.

Mounted FileSystems may need to be removed from a *FileManager*.

3.1.3.3.3.5.2.2 Synopsis.

```
void unmount(in string mountPoint) raises( NonExistentMount );
```

3.1.3.3.3.5.2.3 Behavior.

The *unmount* operation shall remove a mounted *FileSystem* from the *FileManager* whose mounted name matches the input mountPoint name.

3.1.3.3.3.5.2.4 Returns.

This operation does not return any value.

3.1.3.3.3.5.2.5 Exceptions/Errors.

The *unmount* operation shall raise the NonexistentMount exception when the mountPoint does not exist.

3.1.3.3.3.5.3 *getMounts*.

3.1.3.3.3.5.3.1 Brief Rationale.

File management user interfaces may need to list a *FileManager's* mounted *FileSystems*.

3.1.3.3.3.5.3.2 Synopsis.

```
MountSequence getMounts();
```

3.1.3.3.3.5.3.3 Behavior.

The *getMounts* operation shall return a sequence of Mount structures that describe the mounted *FileSystems*.

3.1.3.3.3.5.3.4 Returns.

The *getMounts* operation returns a sequence of Mount structures.

3.1.3.3.3.5.3.5 Exceptions/Errors.

This operation does not raise any exceptions.

3.1.3.3.3.5.4 File System Operations.

The system may support multiple *FileSystem* implementations. Some *FileSystems* will correspond directly to a physical file system within the system. The *FileManager* interface shall support a federated, or distributed, file system that may span multiple *FileSystem* components. From the client perspective, the *FileManager* may be used just like any other *FileSystem* component since the *FileManager* inherits all the *FileSystem* operations.

The *FileManager's* inherited *FileSystem* operations behavior shall implement the requirements of the *FileSystem* operations against the mounted file systems. The *FileSystem* operations ensure that the filename/directory arguments given are absolute pathnames relative to a mounted *FileSystem*. The *FileManager's FileSystem* operations shall remove the *FileSystem* mounted name from the input fileName before passing the fileName to an operation on a mounted *FileSystem*.

The *FileManager* shall use the mounted *FileSystem* for *FileSystem* operations based upon the mounted *FileSystem* name that exactly matches the input fileName to the lowest matching subdirectory.

3.1.3.3.3.5.5 *query*.

3.1.3.3.3.5.5.1 Brief Rationale.

The inherited *query* operation provides the ability to retrieve the same information for a set of file systems.

3.1.3.3.3.5.5.2 Synopsis.

```
void query(inout Properties fileSystemProperties) raises(
UnknownFileSystemProperties );
```

3.1.3.3.3.5.5.3 Behavior.

The *query* operation shall return the combined mounted file systems information to the calling client based upon the given input *fileSystemProperties'* IDs. As a minimum, the *query* operation shall support the following input *fileSystemProperties* IDs:

1. SIZE - a property item ID value of "SIZE" will cause the *query* operation to return the combined total size of all the mounted file system as an unsigned long long property value.

2. AVAILABLE_SPACE - a property item ID value of "AVAILABLE_SPACE" will cause the *query* operation to return the combined total available space (in octets) of all the mounted file system as unsigned long long property value.

3.1.3.3.3.5.5.4 Returns.

This operation does not return any value.

3.1.3.3.3.5.5.5 Exceptions/Errors.

The *query* operation shall raise the UnknownFileSystemProperties exception when the input *fileSystemProperties* parameter contains an invalid property ID

3.1.3.3.4 *Timer*.

No SCA-mandated *Timer* interfaces have been defined at this time.

3.1.3.4 Domain Profile.

The hardware devices and software components that make up an SCA system domain are described by a set of files that are collectively referred to as a Domain Profile. These files describe the identity, capabilities, properties, inter-dependencies, and location of the hardware devices and software components that make up the system. All of the descriptive data about a system is expressed in the XML vocabulary. For purposes of this SCA specification, the elements of the XML vocabulary have been based upon the OMG's CORBA Components specification (orbos/99-07-01). [Note: At the time of this writing, 99-07-01 is a draft standard].

The types of XML files that are used to describe a system's hardware and software assets are depicted in Figure 3-34. The XML vocabulary within each of these files describes a distinct aspect of the hardware and software assets.

Domain Profile files shall use the format of the Document Type Definitions (DTDs) provided in Appendix D. DTD files are installed in the domain and shall have ".dtd" as their filename extension. All XML files shall have as the first two lines as an XML declaration (?xml) and a document type declaration (!DOCTYPE). The XML declaration specifies the XML version and whether the document is standalone. The document type declaration specifies the DTD for the document. Example declarations are as follows:

-"<X?xml version="1.0" standalone="no?>"

-"<!DOCTYPE softwareassembly SYSTEM "softwareassembly.2.0.dtd">"

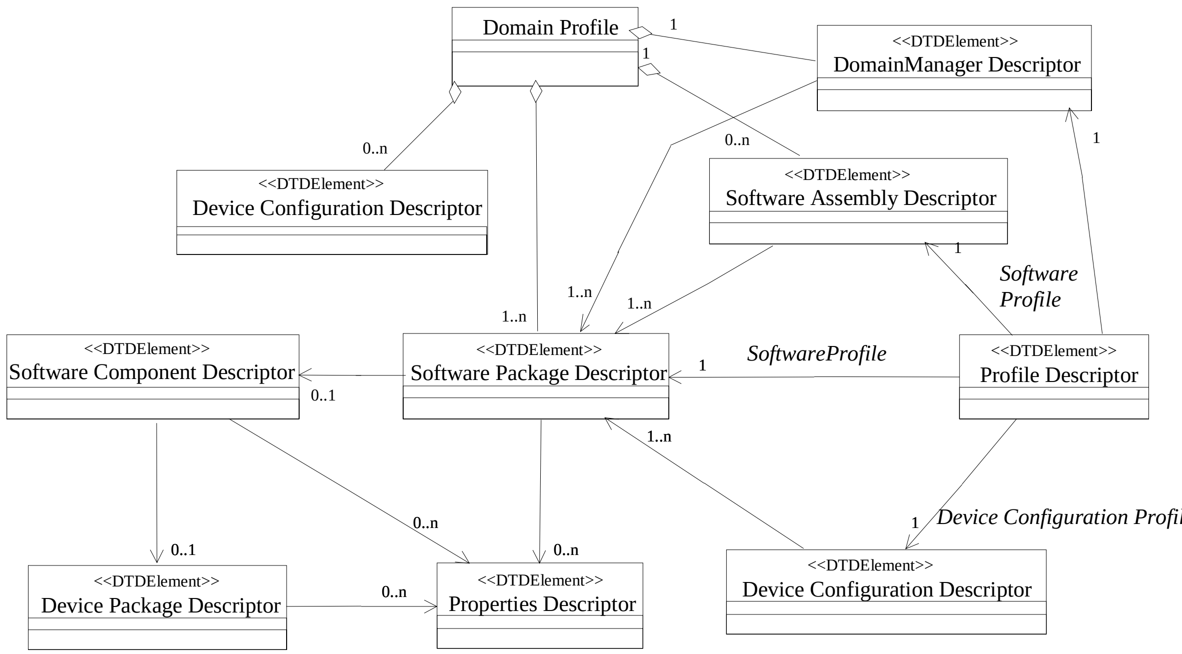

Figure 3-34. Relationship of Domain Profile XML File Types

3.1.3.4.1 Software Package Descriptor.

A Software Package Descriptor (SPD) identifies a software component implementation(s). A Software Package Descriptor file shall have a ".spd.xml" extension. General information about a software package, such as the name, author, property file, and implementation code information and hardware and/or software dependencies are contained in a Software Package Descriptor file.

3.1.3.4.2 Software Component Descriptor.

A Software Component Descriptor (SCD) contains information about a specific SCA software component (*Resource*, *ResourceFactory*, *Device*). A Software Component Descriptor file shall have a ".scd.xml" extension. A Software Component Descriptor file contains information about the interfaces that a component provides and/or uses. A Software Component Descriptor for a *Device* type has a reference to Device Package Descriptor file.

3.1.3.4.3 Software Assembly Descriptor.

A Software Assembly Descriptor (SAD) contains information about the components that make up an application. The *ApplicationFactory* uses this information when creating an application. A Software Assembly Descriptor file shall have a ".sad.xml" extension.

3.1.3.4.4 Properties Descriptor.

A Property File contains information about the properties applicable to a software package or a device package. A Properties File shall have a ".prf.xml" extension. A Properties File contains information about the properties of a component such as configuration, test, execute, and allocation types.

3.1.3.4.5 Device Package Descriptor.

A Device Package Descriptor (DPD) identifies a class of a device (as described in Section 4). A Device Package Descriptor File shall have a ".dpd.xml" extension. A Device Package

Descriptor also has Properties that define specific properties (capacity, serial number, etc.) for this class of device.

3.1.3.4.6 Device Configuration Descriptor.

A Device Configuration Descriptor (DCD) contains information about the children *Devices* for a *Device*, how to find the *DomainManager*, and the configuration information (*Log*, *FileSystems*, etc.) for a *Device*. A Device Configuration Descriptor file shall have a ".dcd.xml" extension.

3.1.3.4.7 Profile Descriptor

A Profile Descriptor contains an absolute file name for either a Software Package Descriptor, Software Assembly Descriptor, or a Device Configuration Descriptor. The Profile Descriptor is derived from the *Application*, *ApplicationFactory*, and *Device* attributes.

3.1.3.4.8 *DomainManger* Configuration Descriptor.

A *DomainManager* Configuration Descriptor (DMD) contains configuration information for the *DomainManager*. A *DomainManager* Configuration Descriptor file shall have a ".dmd.xml" extension.

3.1.3.5 Core Framework Base Types.

The CF Base Types are the underlying types used in the CF interfaces.

3.1.3.5.1 Data Type.

This type is a CORBA IDL struct type, which can be used to hold any CORBA basic type or static IDL type. The id attribute indicates the kind of value and type (e.g., frequency, preset, etc.). The id can be an UUID string, an integer string, or a name identifier. The value attribute can be any static IDL type or CORBA basic type.

```
struct DataType {
 string id;
 any value;
};
```

3.1.3.5.2 DeviceSequence.

The CF DeviceSequence type defines an unbounded sequence of *Devices*. The IDL to Ada mapping has a problem with self-referential interfaces. To get around this problem, the interface Device forward declaration has been created and this type has been moved outside of the *Device* interface.

```
typedef sequence <Device> DeviceSequence;
```

3.1.3.5.3 FileException.

The CF FileException indicates a file-related error occurred. The message provides information describing the error. The message can be used for logging the error. The error code describes the type of error message.

```
exception FileException {unsigned short errorCode; string msg;};
```

3.1.3.5.4 InvalidFileName.

The InvalidFileName exception indicates an invalid file name was passed to a file service operation. The message provides information describing why the filename was invalid.

```
exception InvalidFileName {string msg;};
```

3.1.3.5.5 InvalidObjectReference.

The InvalidObjectReference exception indicates an invalid CORBA object reference error.

```
exception InvalidObjectReference {string msg;};
```

3.1.3.5.6 InvalidProfile.

The InvalidProfile exception indicates an invalid profile error.

```
exception InvalidProfile {};
```

3.1.3.5.7 OctetSequence.

This type is a CORBA unbounded sequence of octets.

```
typedef  sequence <octet> OctetSequence;
```

3.1.3.5.8 Properties.

The properties is a CORBA IDL unbounded sequence of CF Data Type(s), which can be used in defining a sequence of name and value pairs.

```
typedef sequence <DataType> Properties;
```

3.1.3.5.9 StringSequence.

This type defines a sequence of strings.

```
typedef sequence <string> StringSequence;
```

3.1.3.5.10 UnknownProperties.

The UnknownProperites exception indicates a set of properties unknown by the component.

```
exception UnknownProperties {Properties invalidProperties };
```

3.1.3.5.11 DeviceAssignmentType.

DeviceAssignmentType defines a structure that associates a component with the *Device* upon which the component must execute.

```
Struct DeviceAssignmentType
{
string componentID;
string assignedDeviceID;
}
```

3.1.3.5.12 DeviceAssignmentSequence.

The IDL sequence, CF DeviceAssignmentSequence, provides an unbounded sequence of 0..n CF DeviceAssignmentTypes.

```
Typedef sequence <DeviceAssignmentType> DeviceAssignmentSequence;
```

3.2 APPLICATIONS.

Applications are programs that perform the functions of a specific SCA-compliant product. They must meet the requirements of a procurement specification and are not defined by the SCA except as they interface to the OE.

3.2.1 General Application Requirements.

3.2.1.1 OS Services.

Applications shall be limited to using the OS services that are designated as mandatory in the SCA AEP as specified in section 3.1.1.

Applications shall perform file access through the CF *File* interfaces. Application file names shall not exceed 40 characters.

To ensure controlled termination, applications shall have a signal handler installed for the POSIX-defined SIGQUIT signal.

3.2.1.2 CORBA Services.

Applications shall be limited to using CORBA and CORBA services as specified in section 3.1.2. Use of Naming Services per 3.1.2.2.1 is optional; if Naming Service is not used, applications shall include stringified IORs in their Software Profile. The use of Log interface per section 3.1.2.3.3 is optional if informational messages are not logged.

3.2.1.3 CF Interfaces.

Applications shall implement the CF interfaces as specified in section 3.1.3.1 using the corresponding IDL in Appendix C. The following exceptions to the use of CF interfaces are allowed:

1. The use of *ResourceFactory* per section 3.1.3.1.7 is optional.

The *TestableObject runTest* operation (3.1.3.1.3.5.1), *Resource stop* operation (3.1.3.1.6.5.1), and *Resource start* operation (3.1.3.1.6.5.2) are not called at start-up.

Each application process that uses Naming Service shall support the name parameters passed by the *DeviceManager* (/ DomainName / NodeName / [other context sequences] / ComponentName_UniqueIdentifier) (see also 3.1.3.2.2.5.1.3). The application shall place this registration underneath the last naming context passed to the application. (In the naming parameter string, each "slash" (/) represents a separate naming context.) Application’s executable components shall accept arguments of the form described in 3.1.3.2.6.5.1.3.

Applications' components and *DeviceManagers* shall be provided with Domain Profile files per 3.1.3.4.

3.2.2 Application Interfaces.

Applications consist of one to many components. These components may be CORBA-capable or not CORBA-capable components. For CORBA-capable components, in addition to supporting the CF Base Application interfaces, the component can implement and use component-specific interfaces for data and/or control. Interfaces provided by a component shall be described in a Software Component Descriptor file as provides ports. Interfaces used by a component shall be described in a Software Component Descriptor file as uses ports.

An application may have other external interfaces besides the *Application* interface. The optional external interfaces for an application are the components' ports referenced in the application's SAD externalports element. The application's external interfaces shall be visible and defined as described herein if:

1. the application provides a service that is used by more than one application, or
2. the service user requires the interface to be common across access service implementations (e.g., HCI).

Figure 3-35 and Figure 3-36 depict push and pull interfaces using the basic CORBA types and the CF Properties for the control type. If used, push and pull interfaces shall be implemented using the IDL in Appendix C. The push interfaces are based on up on the API Packet Building Block (see the API Supplement to the SCA Specification).

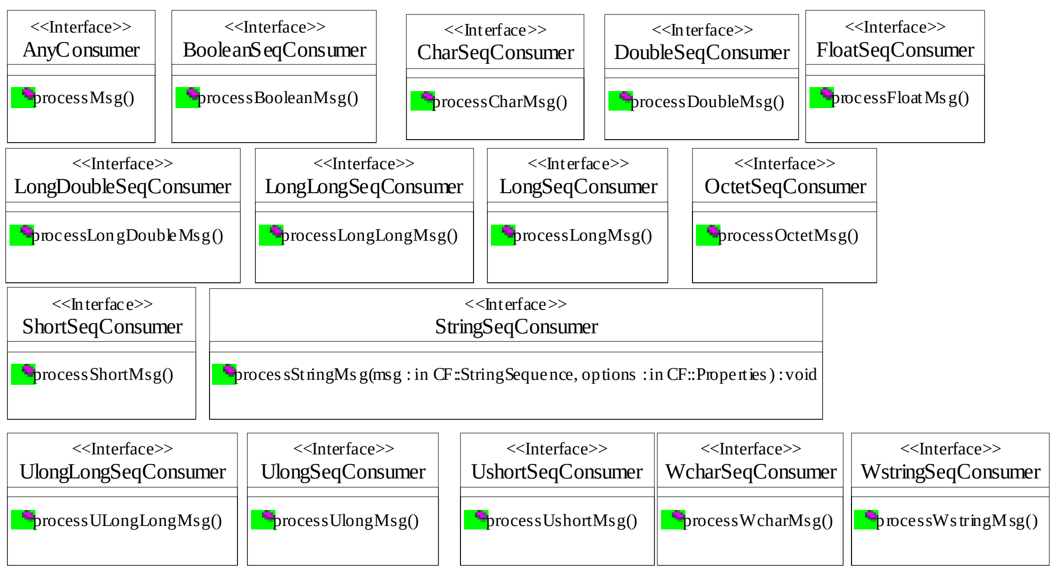

Figure 3-35. PushPort Data Interfaces

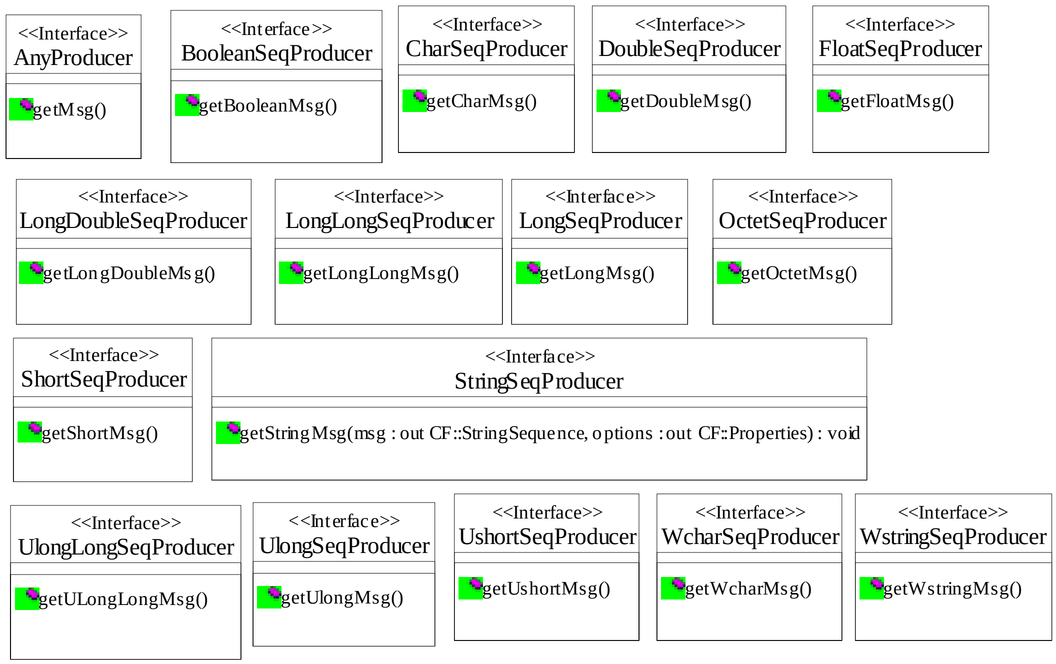

Figure 3-36. PullPort Data Interfaces

3.2.2.1 Utility Applications.

3.2.2.1.1 Installer Utility.

Installer is a generic name given to utility applications used for installing and uninstalling devices and components within the system. Installers shall use the *DomainManager*, *DeviceManager*, and *FileManager* interfaces for those operations. Installers can reside inside or outside the system.

Figure 3-37 and Figure 3-38 illustrate device and software installations using the CF.

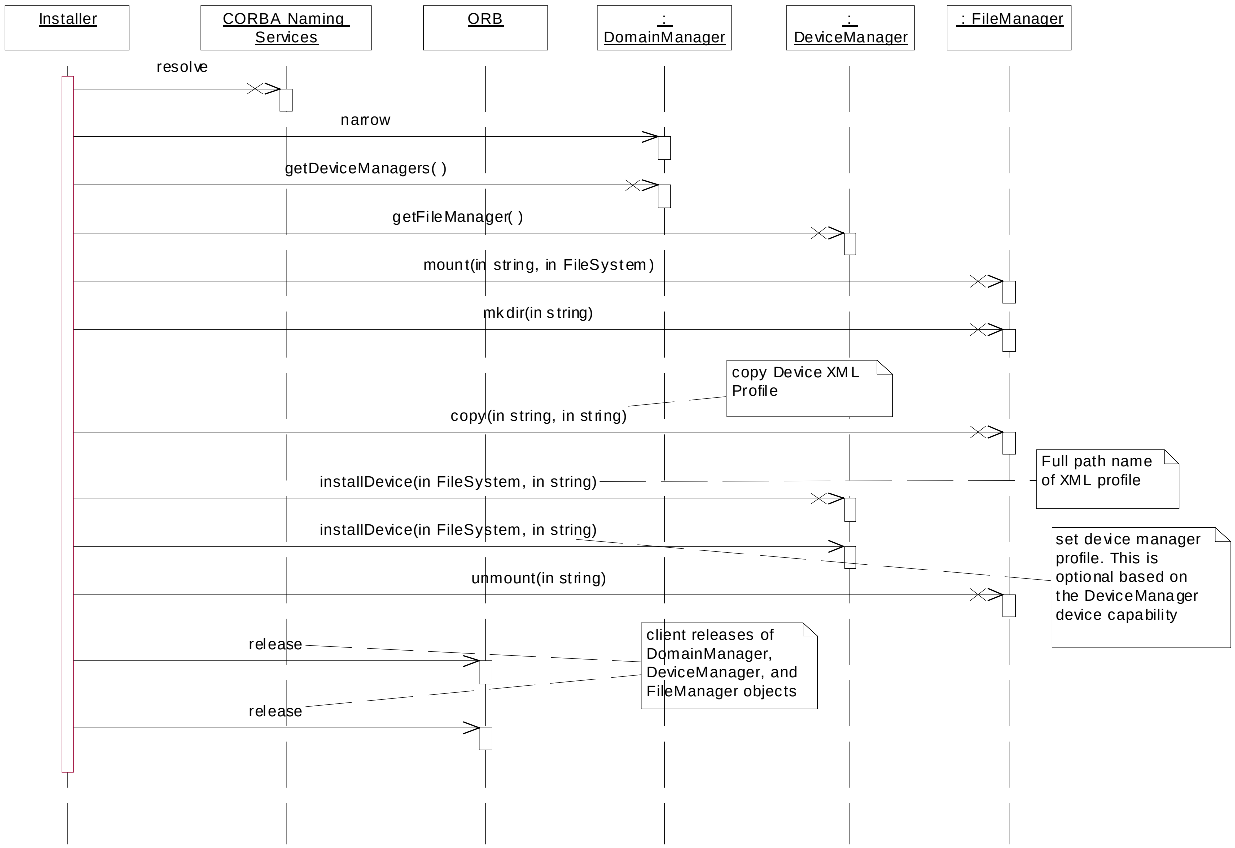

Figure 3-37. Device Installation Sequence Diagram

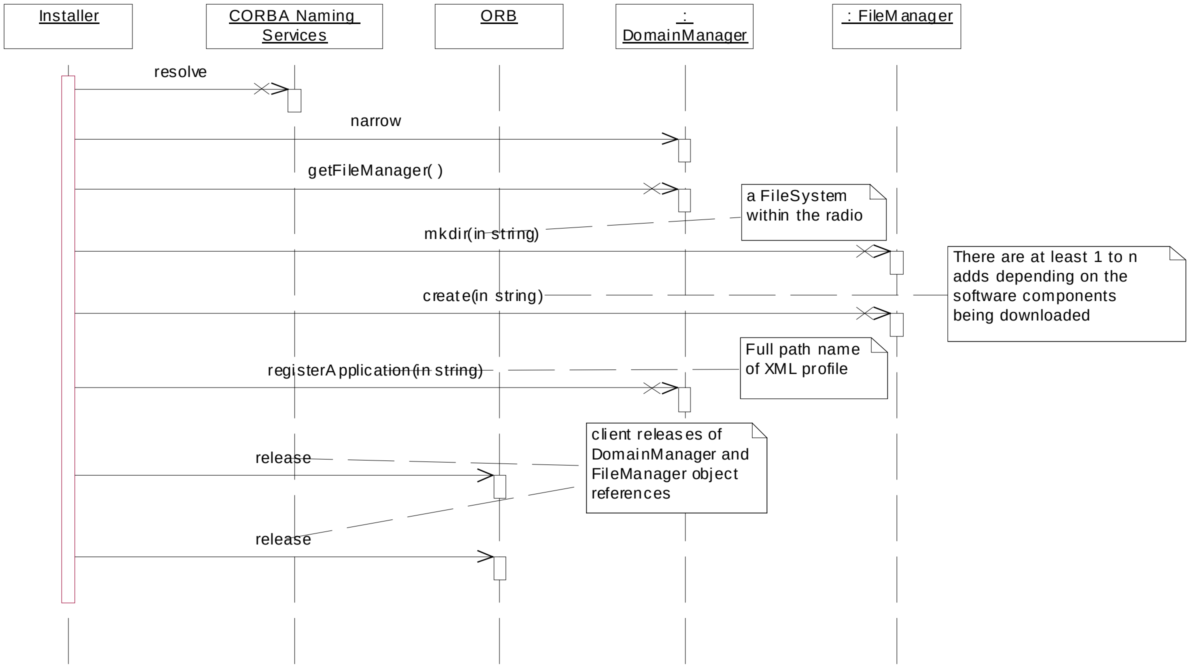

Figure 3-38. Software Installation Sequence Diagram

3.2.2.2 Service APIs.

Service APIs provide definition and standardization of common functionality and interfaces for use by SCA applications (e.g. waveforms). Services include Network Services, Security Services, and I/O Services. Each Service API is defined by a Service Definition and Transfer Mechanism. The API Supplement to the SCA Specification provides details and requirements for Service APIs.

3.2.2.2.1 Service Definitions.

SCA-compliant Service Definitions consist of APIs, behavior, state, priority and additional information that provide the contract between the Service Provider and the Service User. IDL is used to define the interfaces for Service Definitions to foster reuse and interoperability. IDL provides a method to inherit from multiple interfaces to form a new Service Definition.

3.2.2.2.2 API Transfer Mechanisms.

A Transfer Mechanism provides the communication between a service provider and a service user that may be co-located or distributed across different processors. Figure 3-39 shows the standard and alternate transfer mechanism structure for APIs.

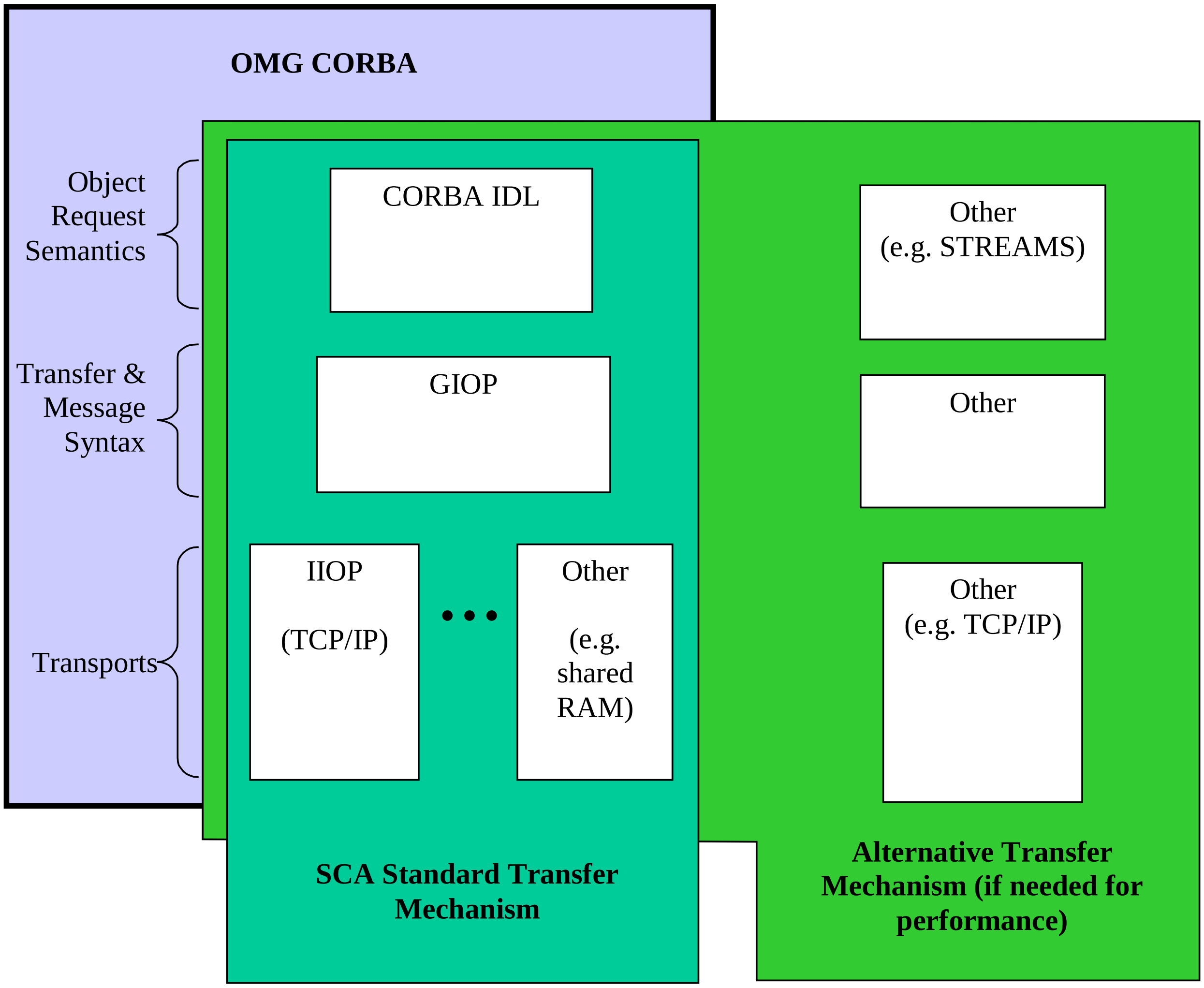

Figure 3-39. Standard and Alternate Transfer Mechanism

3.3 LOGICAL DEVICE.

A logical *Device* is a software proxy for a hardware device(s). Each hardware device used by an application Resource components shall have an associated logical *Device* interface. Logical *Device* interfaces include *Device, LoadableDevice, ExecutableDevice, and AggregateDevice.* The logical *Device* interfaces are depicted in Figure 3-40.

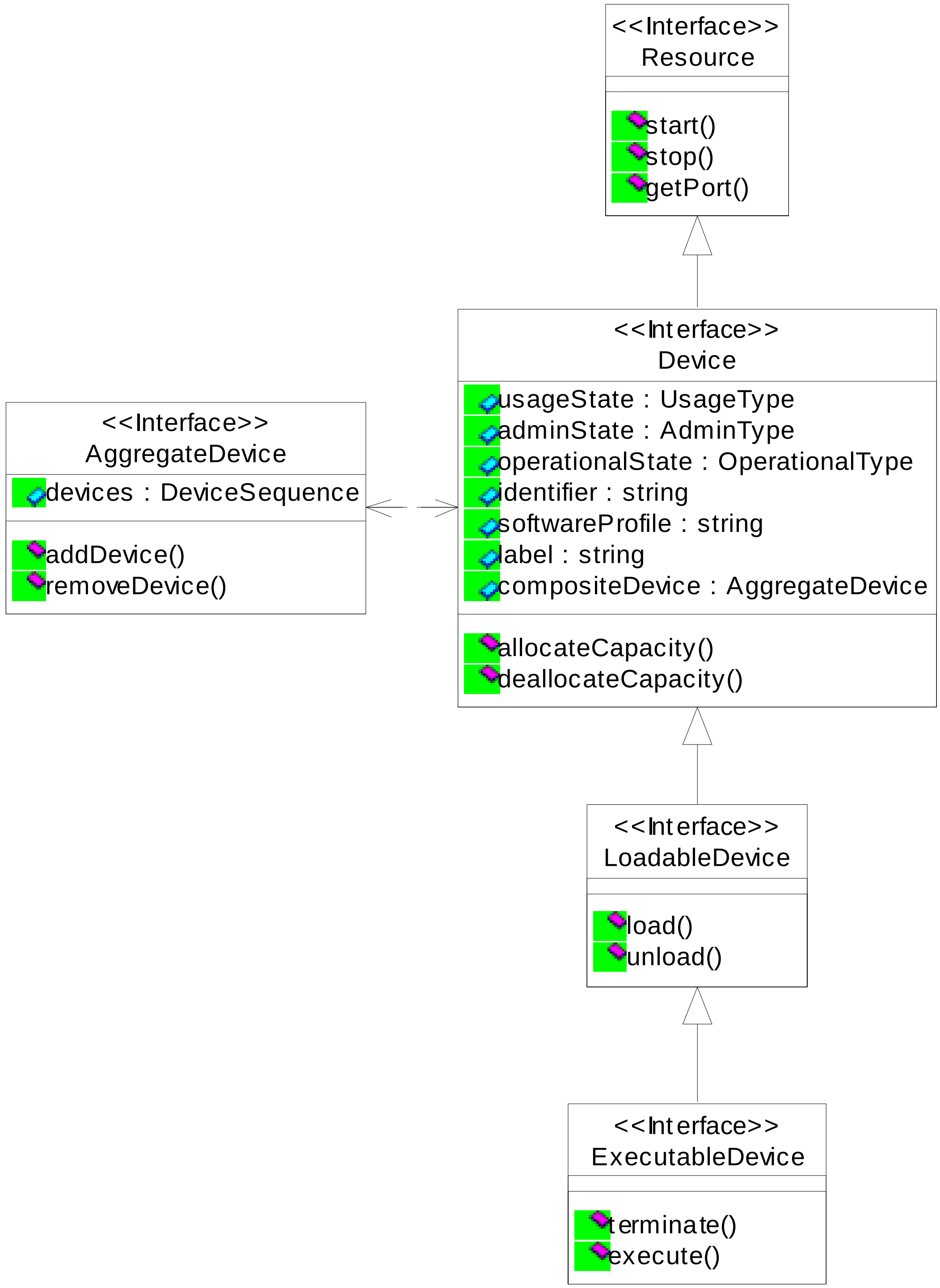

Figure 3-40. ***Device and DeviceManager*** **Interfaces Relationships**

3.3.1 OS Services.

Logical *Devices* are not restricted to using the services designated as mandatory by the SCA AEP as specified in 3-1.

A logical *Device's* executable parameters shall accept arguments of the form described in 3.1.3.2.6.5.1.3.

A logical *Device* shall accept the executable parameters as specified described in 3.1.3.2.8.5.

3.3.2 CORBA Services.

Logical *Devices* shall be limited to using CORBA and CORBA services as specified in section 3-2.

3.3.3 CF Interfaces.

A logical *Device* implements one of the following CF interfaces: *Device, LoadableDevice* or *ExecutableDevice*.

In addition to the requirements stated in the *Device* interface (section 3.1.3.2.4), a logical *Device* has the requirements as stated in the *Resource*, *PropertySet*, *Lifecycle*, *Port*, *PortSupplier* and *TestableObject* interfaces.

A logical *Device* shall register itself with a *DeviceManager* using the executable *DeviceManager* IOR parameter per 3.1.3.2.8.5.

An aggregated logical *Device* shall add itself to a composite *Device* using the executable Composite *Device* IOR parameter per 3.1.3.2.8.5.

The executable parameters (PROFILE_NAME, COMPOSITE_DEVICE_IOR, DEVICE_ID and DEVICE_LABEL) as described in 3.1.3.2.8.5 shall be used to set the *Device's* softwareProfile, compositeDevice, identifier, and label attributes.

A *Device* that has other *Devices* associated with it shall provide a "provides" port that implements the *AggregateDevice* interface. The "provides" port name shall be named "CompositeDevice".

Additional service APIs and their ports beyond the CF adhere to the requirements as described in section 3.2.2.2.

3.3.4 Profile

Each logical *Device* shall have a SPD, SCD, DPD, and one or more Properties Descriptors as described in section 3.1.3.4. For each logical *Device*, allocation properties shall be defined in its referenced SPD's property file.

3.4 GENERAL SOFTWARE RULES.

This section identifies those rules and recommendations specific to the Software Architecture that are not specifically addressed elsewhere in this specification.

3.4.1 Software Development Languages.

3.4.1.1 New Software.

Software developed for an SCA-compliant product shall be developed in a standard higher order language, except at provided below, for ease in processor portability. The goal of new development should be to provide SW that is independent from platform and environment details, ensuring minimal portability issues.

An exception is allowed to this requirement, if there are program performance requirements that require the use of assembly language programming.

3.4.1.2 Legacy Software.

Legacy software is not required to be rewritten in a standard higher order language. Legacy software shall be interfaced to the core framework in accordance with this specification.

4 HARDWARE ARCHITECTURE DEFINITION

This section describes the methodology of using the SCA as the basis for partitioning the Hardware (HW) Architecture in terms of an Object-Oriented approach. This Object-Oriented approach describes a hierarchy of hardware class and subclass objects that represent the architecture. Characteristics, or attributes, associated with each hierarchical class form the domain independent basis for the definition of each physical hardware device. Section 4.5 specifies the hardware requirements.

4.1 BASIC APPROACH.

The definition of the HW Architecture consists of a set of HW classes that are common across all domains. The top-level hardware classes correspond with top-level hardware functions. These top-level HW classes are further refined into subclasses that correspond with lower-level hardware functions. The attributes associated with these classes and/or subclasses describe the individual class or subclass contributions to system features and capabilities.

During implementation, this hardware class structure can be used to describe the hardware implementation in accordance with procurement specifications. This object-oriented approach enables a consistent application of the HW architecture (classes and rules) across the various domains (i.e., Handheld, Dismounted, Vehicular, Airborne, and Maritime/Fixed).

Attributes and the HW class structure will potentially have multiple users over the lifetime of each hardware module. Initially, when the radio system engineer is designing a radio system, class attributes provide a place to sort top level requirements, either by direct allocation or by analysis and allocation. After physical partitioning is performed, the attributes outline HW module(s) specification(s). The hardware designer, through the module specifications, in effect, uses the attributes to characterize the design of the modules.

Software applications also become users of HW attributes. The attributes are reported to the *DomainManager* through the Device Profiles. As software applications become more sophisticated, they will become increasingly dependent upon HW attributes, used potentially both as variables or in software dependency checks in the applications.

4.2 CLASS STRUCTURE.

Class structure is the hierarchy that depicts how object-oriented classes and subclasses are related. The SCA hardware class structure identifies functional elements that in turn are used in the creation of physical system elements (HW devices). Using this object-oriented approach, devices "inherit" from the class structure and share common physical and interface attributes, thus making it easier to identify and compare device interchangeability. (In this use, the term “inherit” simply means that attributes at a higher class-level are common with all the subclasses. In the following figures, this feature is shown by a hollow arrow, the UML symbol for “generalization”.)

Hardware devices represent physical implementations whose attributes are assigned specific values. In this sense, the attributes define domain-neutral class objects (abstract classes) and the values of the class attributes then place specific requirements on the implementation. HW devices inherit common attributes via the hardware class structure. Devices can then be developed to satisfy procurement-specific requirements. All hardware devices will have values assigned to the class attributes. (The attributes shown in the figures in this section are representative of the attributes associated with the respective classes and are provided for illustrative purposes.)

4.2.1 Top Level Class Structure.

The top-level *SCA-Compliant Hardware* class defines the system procurement-associated attributes such as maintainability and availability requirements, as well as, physical, environmental and device registration parameters. (Reference Figure 4-1.) The *Chassis* class has unique physical, interface, platform power and external environment attribute values that are related to external factors rather than individual modules within the chassis. The *HWModule(s)* class represents a wide variety of SCA-compliant physical hardware that inherits attributes from *the SCA-Compliant Hardware* superclass. Subclasses of *HWModule(s)* inherit all its attributes, including those shown in Figure 4-2. Stereotypes, indicated by enclosure in double brackets (<<stereotype>>), are included in the class diagrams to better group and manage attribute labels and titles. The stereotypes are generally associated with particular users of the attributes. The <<Registration>> stereotype attributes are those that become part of a Device Profile as reported through a Device Package Descriptor file. All other stereotypes indicate attributes that, when reported, become part of the Device Profile as reported through a Property File.

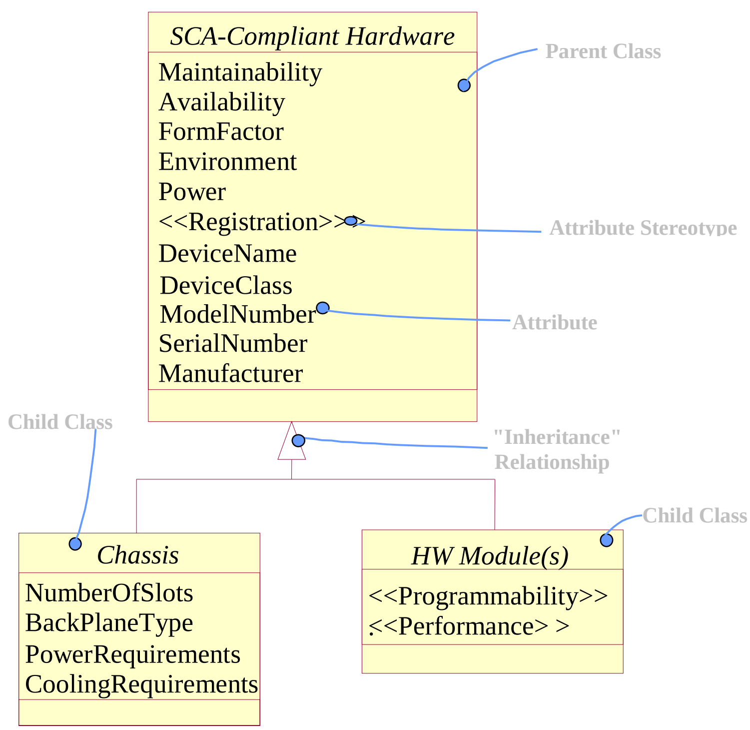

Figure 4-1. Top Level Hardware Class Structure

The *Chassis* subclass includes the attributes of number of module slots, form factor, back plane type, platform environmental, power and cooling requirements. The *HWModule(s)* class is the parent to all module sub-classes and provides the basic attributes that are inherited by all hardware modules. As the class structure hierarchy extends from the more general top level down into the more specific lower levels, each subclass inherits the attributes of all the preceding hierarchy of classes. Module compatibility can be ascertained by comparing appropriate instantiated attributes.

4.2.2 *HWModule(s)* Class Structure.

The JTRS concepts of hardware reuse, extendibility and expandability dictate a modular implementation approach. The hardware architecture presents two very distinct module types. The first type contains software intensive processing elements (i.e., Digital Signal Processor (DSP) modules and General Purpose Processor (GPP) cards), while the second type contains non-programmable functionality (such as RF elements). As programmable capability and programmable hardware technologies evolve, functionality will migrate from totally embedded hardware towards more software intensive applications of the hardware functions.

There is a blurring of hardware/software functionality as systems are implemented. Functions are realized from a combination of both hardware embedded functions and software functions. Thus the *HWModule(s)* class framework shown in figure 4-2 includes functional classes that are strictly programmable in nature (*Processor*) and others that have embedded functionality. This provides the framework necessary to construct the elements for a software programmable radio.

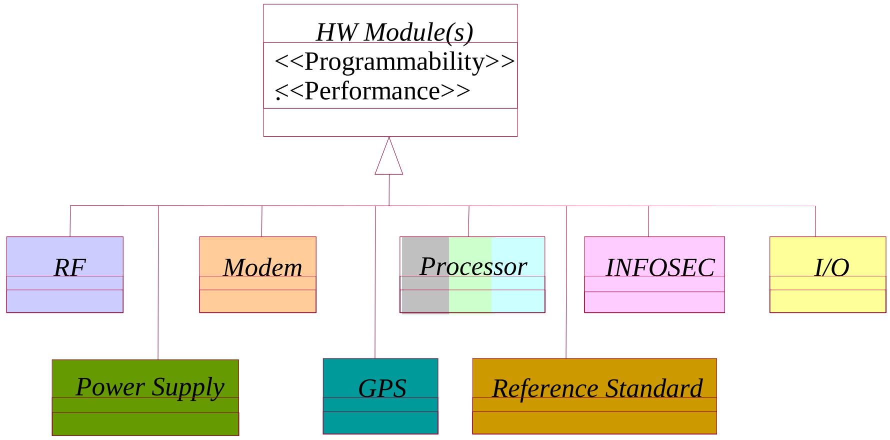

Figure 4-2. Hardware Module Class Structure

The hardware class structure is expandable through the addition of new classes or through the addition of new attributes to existing classes to allow for future growth capabilities. Stereotypes, indicated by enclosure in double brackets (<<stereotype>>), are included in the class diagrams to better group and manage attribute labels and titles.

4.2.3 Class Structure with Extensions.

Each hardware class can be extended further to provide additional attribute granularity. This methodology provides both a formalized structure for hardware definition and the inherent flexibility needed to allow for evolving requirements as well as hardware and software capabilities.

4.2.3.1 *RF* Class Extension.

The subclasses in figure 4-3 extend the *RF* class hierarchy. These subclasses relate to the typical range of RF hardware devices such as, Antennas, Receivers, Exciters, and Power Amplifiers. As with all HW subclasses, the attributes contained within these RF subclasses attempt to encapsulate the functionality that can be used to describe the unique mix of features and capabilities of the associated hardware device.

Cosite performance considerations place a special burden on the *RF* class. The intelligent management of cosite performance requires monitoring and control of many of the RF subclass modules. The hardware architecture supports cosite operation in two ways. First, there is a cosite sub-class. This encapsulates the hardware that is specifically provided for cosite operation. Second, a <<CositePerformance>> stereotype groups those attributes useful for a cosite manager application. Such an application, while not part of the architecture itself, is an implementation-specific capability to coordinate RF assets.

Antennas have historically been passive elements typically attached to the structure that houses the communications system. While remaining very domain and platform unique, technology growth continually improves the capabilities that can be performed in the communications system 'front end', necessitating the inclusion of antennas in the core of JTRS. "Smart" antennas include receive, transmit, and cosite mitigating elements, blurring the functional separation lines. For this reason and because of the key role that antenna systems play in cosite management, “Antenna” is incorporated in the class structure as an RF subclass.

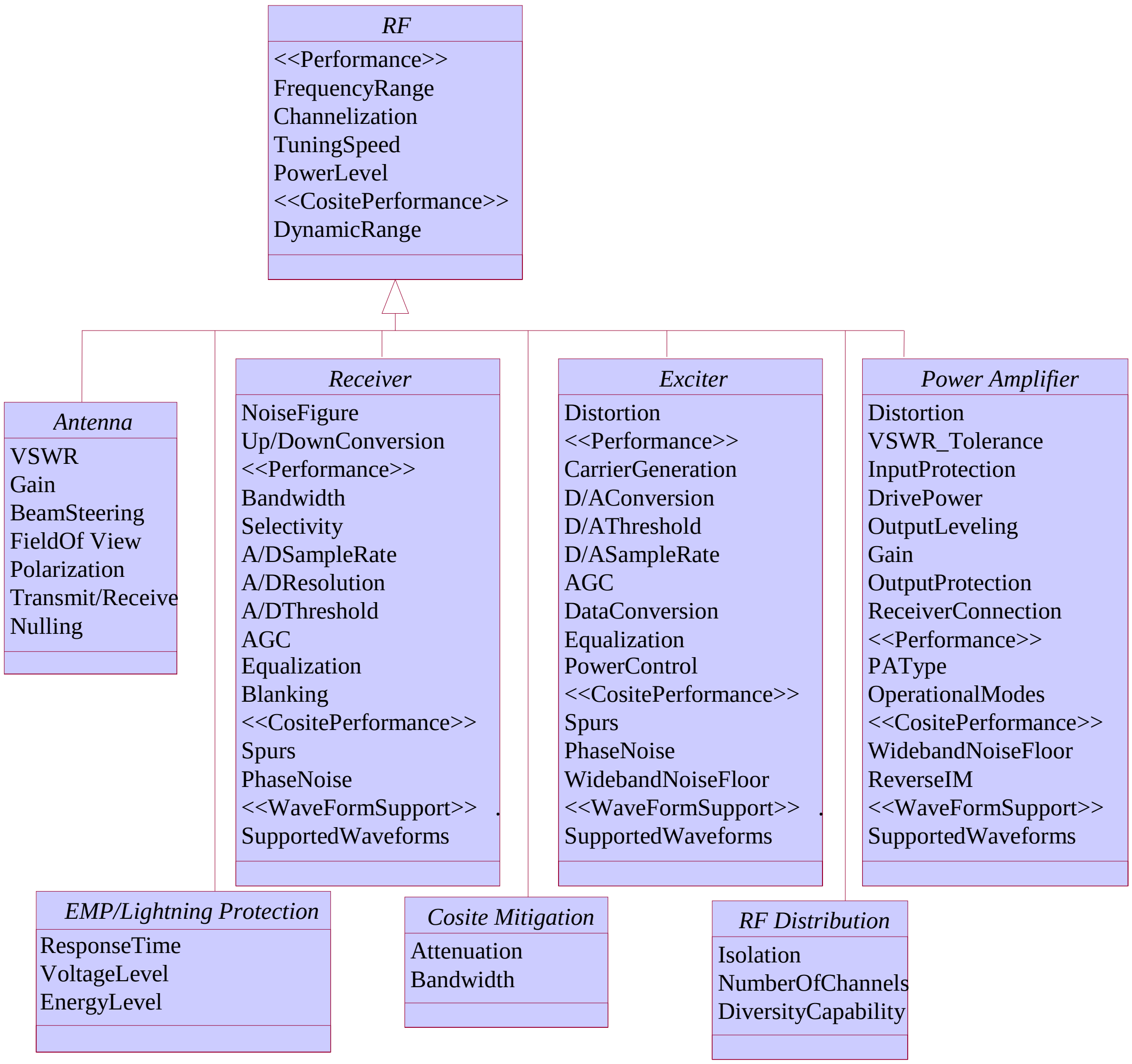

Figure 4-3. RF Class Extension

4.2.3.2 Modem Class Extension.

The *Modem* class shown in figure 4-4 has subclasses that encapsulate the attributes of modulation and demodulation functions. The *Modem* class also contains attributes that can be used to describe the range of signal processing and data conversion capabilities such as spreading and de-spreading. The <<WaveFormSupport>> stereotype labels the attribute of SupportedWaveforms. This is an attribute indicating specifically what waveforms the modem is capable of performing.

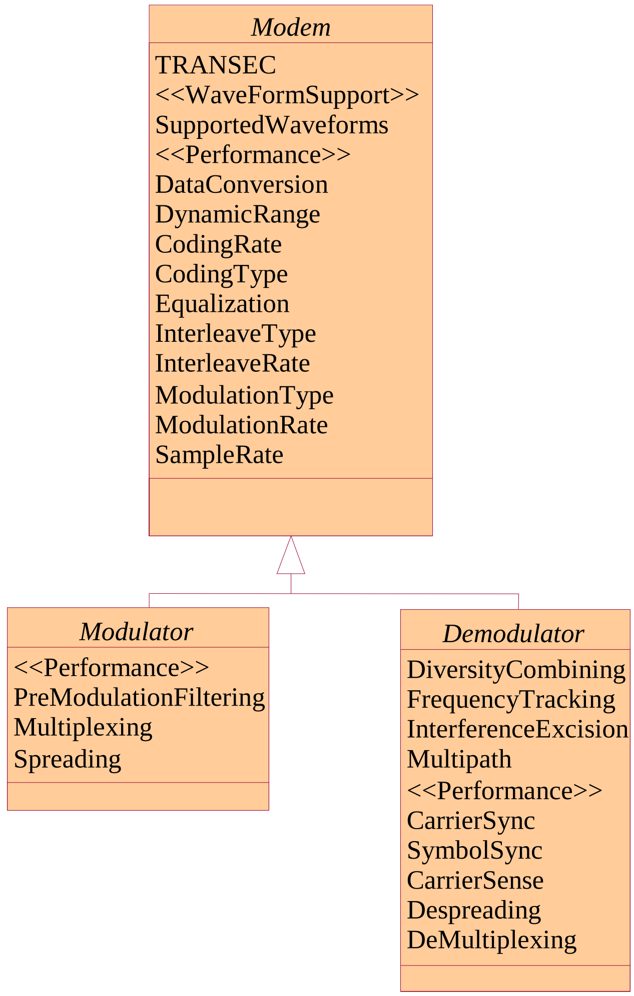

Figure 4-4. Modem Class Extension

4.2.3.3 *Processor* Class Extension.

The *Processor* class shown in figure 4-5 directly supports software operations by providing the processors, memory, and supporting functions. Devices derived from this class include General Purpose Processors, Digital Signal Processors, and extend to modules utilizing programmable logic devices (FPGAs, etc.). The class captures the attributes of processing devices needed by the system resources. This *Processor* class represents the type of hardware that, in itself, essentially has no unique radio-functional capabilities of its own. Its actual use, or personality, is a function of the software that is loaded into and executed on it. It can be envisioned that as processor speeds and software capabilities evolve, this class of hardware will tend to dominate future radio systems while some of the other hardware specific functions will be replaced by processors and software. As this happens, the attributes associated with function and performance will effectively migrate to the software applications that are running on the host processors.

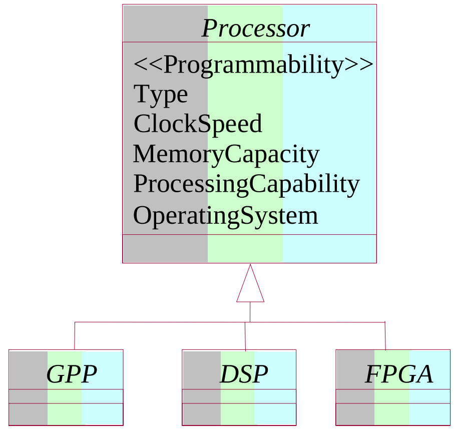

Figure 4-5. Processor Class

4.2.3.4 *INFOSEC* Class.

The *INFOSEC* class provides structure for a hardware device that is described by the type of cryptographic features it supports and certifications for which it has been qualified. Figure 4-6 lists *INFOSEC* class attributes.

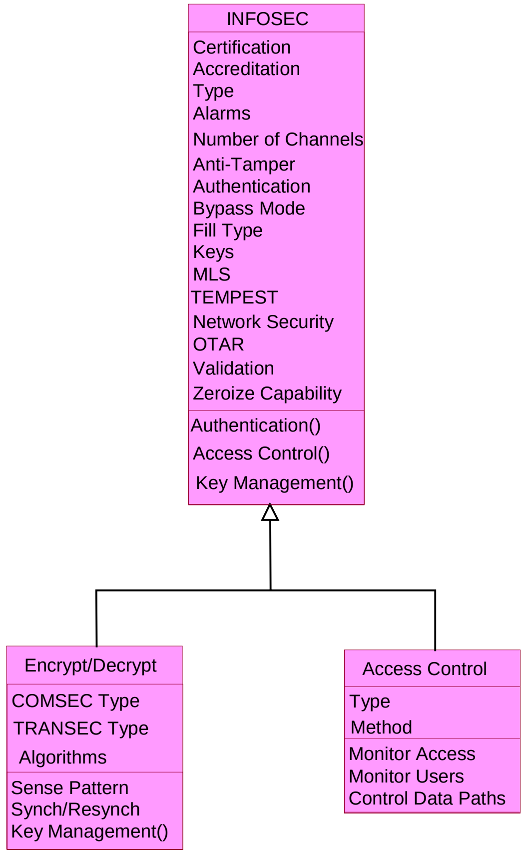

Figure 4-6. INFOSEC Class

4.2.3.5 *I/O* Class Extension.

The *I/O* Class shown in figure 4-7 provides representation for general physical connectivity and is not limited to just user interfaces.

For every hardware device, the critical interfaces are those that are presented to the "outside world". The definition of a critical interface is dependent on the class hierarchy level at which the hardware device is being viewed. For example, if the HW device is a complete radio system, it inherits attributes from the chassis class and its critical interfaces are defined at the chassis physical boundary. Additionally, each module within the radio system has critical interfaces unique to it; and its I/O attributes are inherited from the *I/O* subclass.

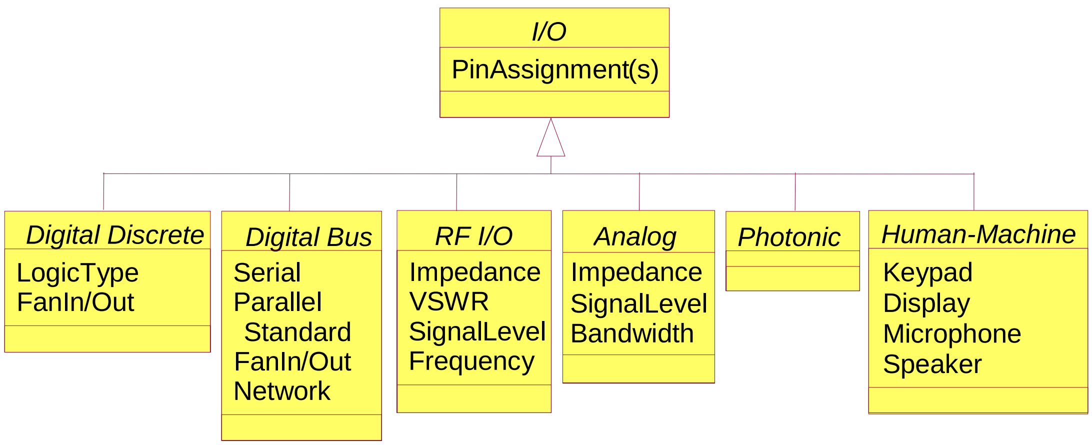

Figure 4-7. I/O Class Extension

4.2.4 Attribute Compostion.

As hardware technology evolves, hardware modules will encompass increased levels of functionality due to higher levels of integration. This will allow more functional hardware classes to be realized within individual physical hardware modules. The function of the individual classes remains the same, but they are physically realized on the same circuit card or module. UML provides the 'composition' relationship to represent this. An example of this is shown in figure 4-8, showing a module that provides receive, transmit, and modulation/demodulation capabilities, and using the hardware class model to illustrate this fusion of capabilities. The resultant attribute list for the module will be composed of the unique mix of features encapsulated by the four hardware classes from which it is composed. Since each of the individual classes inherit attributes from its respective higher level class, the hardware module also inherits from the higher levels.

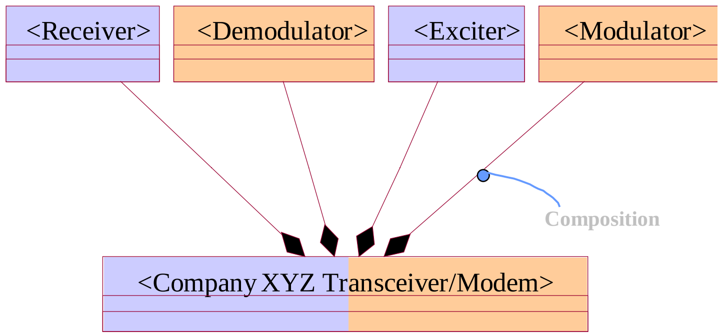

Figure 4-8. Typical Hardware Device Description using the SCA HW Class Structure

4.3 DOMAIN CRITERIA.

As communications systems assume multi-band, multi-channel, and multi-mission capabilities, a dilemma arises. When trying to satisfy the needs of both the small, highly mobile user (Handheld Domain) and the large command center (Maritime/Fixed Domain), it is evident that distinctly different mission and platform constraints exist. Offering the same solution for both extremes is obviously not the optimum – or cost effective – solution for either. The highly-mobile user requires a compact, environmentally-robust terminal containing embedded message processing, sized sufficiently to their needs, but not so large as to meet the intensive filtering/formatting/networking needs of the command center. The command center, on the other hand, requires environmental robustness only to the inhabited level. There are many, real barriers to complete commonality - cost being the largest. The most significant hardware cost-savings potential is the use of COTS standards, technology, and components, where possible. The SCA provides the standard for use of COTS technology, design reuse across products, and an open, well-documented architecture allowing multiple contractors to implement an entire system or only a portion of it.

4.4 PERFORMANCE RELATED ISSUES.

A particular implementation of the SCA can have significant impact on the equipment performance, especially in the case of complex waveforms and multi-channel radios. The areas of cosite performance and system control timing have been identified as two key performance areas for careful consideration. Discussions of the cosite effects and mitigation techniques applicable to the physical implementation of the architecture are in the SRD.

4.5 GENERAL HARDWARE RULES.

Requirements placed on hardware objects by the SCA reflect a balance between the need to support extendibility and interchangeability, and the support of technology growth and domain constraints. The result is a limited set of specific rules (listed below) augmented by implementation guidelines, much of which is in the SRD.

4.5.1 Device Profile.

Each supplied hardware device shall be provided with its associated Domain Profile files as defined in section 3.1.3.4, Domain Profile.

4.5.2 Hardware Critical Interfaces.

4.5.2.1 Interface Definition.

Hardware critical interfaces shall be defined in Interface Control Documents that are available to other parties without restriction. Critical interfaces are those interfaces at the physical boundary of a replaceable device that are required for the operation and maintenance of the device.

4.5.2.2 Interface Standards.

Hardware critical interfaces shall be in accordance with commercial or government standards, unless there are program performance requirements that require a non-standard interface. If so required, the non-standard interface shall be clearly and openly documented to the extent that interfacing or replacement hardware can be developed by other parties without restriction.

4.5.2.2.1 Interface Selection.

In addition to the above, interface selection should consider the availability of supporting products that have wide usage, are available from multiple vendors, and are expected to have long-term support in the industry.

4.5.3 Form Factor.

The form factor of the hardware objects should be, where practical, in accordance with commercial standards.

4.5.4 Modularity.

The partitioning of the hardware architecture into modules should be chosen to allow for ease of upgrade through technology insertion or replacement of modules based on form, fit, and function. Module boundaries are critical interfaces as defined in 4.5.2.1.

5 SECURITY ARCHITECTURE DEFINITION

The security requirements in this section apply to the CF when security is implemented in a JTRS. Additional security requirements, beyond the CF, are in the Security Supplement to the SCA.

5.1 ADDITIONAL CF SECURITY REQUIREMENTS.

5.1.1 *Application.*

The *Application releaseObject* operation shall only disconnect components’ ports that are authorized by an authentication service.

The *Application releaseObject* operation shall request removal of the *Application’s Ports’* access setups from the access control database.

The *Application releaseObject* operation shall log a Security_Alarm event when unable to disconnect components’ ports because authorization was not granted by an authentication service.

Application components' SPD *implementation dependency propertyref* elements shall indicate a dependency to a red or black device (directly or indirectly).

5.1.2 *ApplicationFactory.*

The *ApplicationFactory create* operation shall only create components that are authorized by an authentication service.

The *ApplicationFactory create* operation shall only connect components’ ports together that are authorized by an authentication service.

If port connections between components need to be access-controlled during execution, then the *ApplicationFactory create* operation shall provide an update to the access control database. The *ApplicationFactory create* operation shall provide updates to an access control database for all components ports connections as stated in the application's SAD file.

The *ApplicationFactory* shall log a Security_Alarm event when unable to connect ports or create components because authorization was not granted by an authentication service.

5.1.3 *DomainManager.*

The *DomainManager installApplication* operation shall send the information specified in the Security Supplement to the control/bypass mechanism *Resource* for the black-side components being accessed by red-side components and for red-side components being accessed by black-side components.

The *DomainManager uninstallApplication* operation shall request removal of the application’s information specified in the Security Supplement from the control/status bypass mechanism.

Devices SPD properties shall have an allocation property that indicates a red or black device. Parent *Devices* shall send their child *Devices* information specified in the Security Supplement to the control/status bypass mechanism. A parentless *Device* shall send its information specified in the Security Supplement to the control/status bypass mechanism.

6 COMMON SERVICES AND DEPLOYMENT CONSIDERATIONS

6.1 COMMON SYSTEM SERVICES.

This section will define any common system services that are not part of the CF but are considered part of the SCA. None have been identified at this time.

6.2 OPERATIONAL AND DEPLOYMENT CONSIDERATIONS.

This section will address common interfaces or features necessary to support deployment of SCA-compliant systems in the field. None have been identified at this time.

7 ARCHITECTURE COMPLIANCE

This section defines the criteria for certifying candidate system, hardware, and software application products to this specification.

This specification may be applied to procurement of a multitude of radio products and communication systems. In addition, this specification may also be applied to hardware-only or software-only products that would be hosted on SCA-compliant systems.

7.1 CERTIFICATION AUTHORITY.

The JTRS Joint Program Office (JPO) holds the authority to certify that a candidate product meets the requirements of this specification. This authority may be transferred, in time, to a general standards body.

7.2 RESPONSIBILITY FOR COMPLIANCE EVALUATION.

The responsibility for performing the evaluation of a candidate product's compliance is TBD. This body will determine the test methods and procedures used to establish compliance.

7.3 EVALUATING COMPLIANCE.

Compliance to this specification is defined as meeting all requirements, except as specifically allowed herein. Products submitted as "SCA-Compliant" will be evaluated for compliance in accordance with the test methods and procedures established per section 7.2.

7.4 REGISTRATION.

Documentation of some elements of an SCA implementation, as defined in sections 3 and 4, will be submitted to a Registration Body to be established, initially, by the JTRS JPO.

[The establishment, membership, rules, and operation of Registration Bodies are beyond the scope of the SCA.]

Some elements of an SCA implementation are identified with a UUID. As used in this specification, the UUID is defined by the DCE UUID standard (adopted by CORBA). (OSF Distributed Computing Environment, DCE 1.1 Remote Procedure Call) No centralized authority is required to administer UUIDs (beyond the one that allocates IEEE 802.1 node identifiers [Medium Access Control (MAC) addresses]).